UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

DEPARTAMENTO DE INFORMÁTICA Y SISTEMAS

TESIS DOCTORAL

ACELERADORES VECTORIALES PARA PROCESADORES SUPERESCALARES

FRANCISCA QUINTANA DOMÍNGUEZ

Las Palmas de Gran Canaria, Noviembre del 2001

27/2001-02

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
UNIDAD DE TERCER CICLO Y POSTGRADO

Reunido el día de la fecha, el Tribunal nombrado por el Excmo. Sr. Rector Magfco. de esta Universidad, el/a aspirante expuso esta TESIS DOCTORAL.
Terminada la lectura y contestadas por el/a Doctorando/a las objeciones formuladas por los señores miembros del Tribunal, éste calificó dicho trabajo con la nota de SOBRESALIENTE CUM-LAUDE

Las Palmas de Gran Canaria, a 19 de diciembre de 2001.

El/a Presidente/a: Dr.D. Roberto Moreno Díaz,

El/a Secretario/a: Dr.D. Enrique Fernández García,

El/a Vocal: Dr.D. José María Llabería Griñó,

El/a Vocal: Dr.D. Francisco Tirado Fernández,

El/a Vocal: Dr.D. Emilio López Zapata,

La Doctoranda: Dª. Francisca Candelaria Quintana Domínguez,

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

Departamento de Informática y Sistemas

TESIS DOCTORAL

ACELERADORES VECTORIALES PARA PROCESADORES SUPERESCALARES

Francisca Quintana Domínguez

Las Palmas de Gran Canaria

Noviembre de 2001

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

Departamento de Informática y Sistemas

Tesis doctoral titulada **ACELERADORES VECTORIALES PARA PROCESADORES SUPERESCALARES**, que presenta Dña. Francisca Quintana Domínguez, realizada bajo la dirección de los doctores D. Mateo Valero Cortés y D. Roger Espasa Sans.

Las Palmas de Gran Canaria, Noviembre de 2001

La doctoranda

Los directores

Dña. Francisca Quintana Domínguez

D. Mateo Valero Cortés

D. Roger Espasa Sans

ACKNOWLEDGMENTS

Quiero agradecer a Mateo Valero y Roger Espasa, mis directores de tesis, su guía y su apoyo durante la realización de esta tesis. Mateo, que me aceptó como doctoranda sin conocerme, ha sido más que un director. Siempre me ha aconsejado honradamente y me ha ayudado en todo lo que le he pedido y más. Roger, con quien he discutido hasta la saciedad, ha sido insuperable como compañero de fatigas. Supe que había terminado esta tesis cuando por fin le gané una discusión. A los dos, muchas gracias.

También me gustaría agradecer a todo el personal del Departamento de Arquitetura de Computadores de la UPC lo bien que me han tratado durante estos años. Con ellos he compartido no sólo el lugar de trabajo, sino también muchas sobremesas entretenidas, algunas juergas, días de playa y un montón de buenos momentos. Me gustaría mencionar especialmente a Montse Peirón, mi primera amiga en Barcelona, Roger Espasa y Marta Jiménez, que confiaron tanto en mi como para dejarme su propia casa, Agustín Fernández, Fermín Sánchez, Dolors Royo, Toni Juan, Josep Llosa, David López, Susana Moreno, Anna del Corral, Jesús Corbal, Cristina Coll y tantos otros. Me gustaría aprovechar esta ocasión para agradecer también a Teresa Monreal los buenos momentos vividos mientras compartimos piso en Barcelona. Afortunadamente, el choque inicial no fue un presagio de lo que luego nos divertiríamos.

Este trabajo no habría sido posible sin el soporte técnico extraordinario del Laboratorio de Cálculo del Departamento de Arquitectura de Computadores de la UPC y del Centro Europeo de Paralelismo de Barcelona. Víctor Mora, Oriol Riu, Judit Jiménez, y en general todos los que allí trabajan, son extraordinarios profesionales. Me gustara agradecer al CEDEX que nos hayan permitido acceder al Convex C4 que poseen, y a Mayte Castro, la ayuda prestada, especialmente durante el terrible "efecto 2000".

Me gustaría agradecer también a mis compañeros del Departamento de Informática y Sistemas, en especial los del área de Arquitectura y Tecnología de Computadores, su apoyo durante todo este tiempo. Ricardo Pérez es un maravilloso compañero de despacho que me alegra la vida con su caracter polifacético. A Carmelo Cuenca, amigo de tantos años, tengo que agradecerle que siempre haya creído en mi, y que me haya apoyado.

Muchas otras personas han contribuido también con su apoyo a que esta tesis salga adelante: Alvaro Suárez me dio el empujón para saltar el charco; Kiko de Sande y Paco Almeida me facilitaron mucho los momentos iniciales; Jose López me ha animado mucho con sus correos al principio y con su presencia al final; Miguel Alemán, amigo reciente, me ha alentado durante estos últimos meses.

Mis padres y mis hermanos siempre han estado ahí, apoyándome y animándome a seguir adelante. A ellos les agradezco su cariño y aliento de tantos años. En especial a mi madre, que se ha encargado siempre de esas pequeñas cosas que no se notan, pero que me han hecho la vida más fácil, sobre todo en los últimos tiempos. Finalmente, a Abraham tengo que agradecerle su amor, su paciencia, su apoyo, y muchas cosas más. Y aunque parezca una broma, a Tricky le agradezco que haya servido de excusa para estirar las piernas y meditar un poco. A todos ellos va dedicado este trabajo.

Este trabajo ha sido realizado gracias a la colaboración de: el Gobierno Autónomo de Canarias, la CICYT (proyectos TIC 0429/95, TIC98-0511-C02-01 y TIC2001-0995-C02-01), la Universidad de Las Palmas de Gran Canaria, el Departamento de Informática y Sistemas de la ULPGC, la Fundación Universitaria de Las Palmas, el Departamento de Arquitectura de Computadores de la UPC, el Centro Europeo de Paralelismo de Barcelona (CEPBA) y el CEDEX.

CONTENTS

1

INTRODUCTION

Summary

In this chapter we present the motivations behind this thesis. Current trends in the exploitation of the different sources of parallelism are presented, especially describing the data-level parallelism (DLP) paradigm and its related advantages. This chapter also analyzes different applications of DLP in past and present computers. The relation of this thesis with previous and current research works is presented, and some comments are made about the application characteristics that are suitable for executing in DLP processors.

1.1 MOTIVATIONS

Over the last two decades, microprocessors have enjoyed a continuous increase in performance and attendant reduction in price/performance [Sla96] [Yu96] [Mal95]. Among the different emerging tendencies, superscalar processors have succeeded in different marketplaces [Sit92] [BDHS94] [Hun95] [TGN95] [WPS95] [Yag96] [Chr96] [Kel96] [Pap96]. From the initial two-way execution in HP PA7100 [AAD+93], DEC Alpha [Sit92], Intel Pentium [AA93] and IBM PowerPC [SDC94], much research efforts have been devoted to the development and enhancement of new and existing techniques that can provide better performance. All these techniques deal with topics such as instruction caches [SV87] [CMMP95] [LBCG95] [WOR96] [WO97], branch prediction [McF93] [YMP93] [SJSM96] [Zha96] [JSN98], register file organization [FJC96] [WB96], register renaming [SP94] [GGV98] [JRB+98], the increasing number of instructions issued for execution [SV87] [FS94], dynamic instruction scheduling and out-of-order execution [PJS97] [VM97], and data caches [WO95] (including multilevel caches [JW94], data prefetching [Zha96] [Vei97], non-blocking accesses [Kro81] [CB92], multiported caches [WOR96], and others [JNT97] [SF91]).

All these improvements have allowed walking one step forward in the performance gain path. Current best of breed microprocessors operate at frequencies over 1GHz [Int00] [Kah99] and offer superscalar instruction dispatch, sophisticated branch prediction techniques and support for high performance memory systems, including on-chip second level caches [Int00] [Kah99] [Kes99] [Yag96] [Kum96]. These advances do not come without a price, and all these techniques are getting even more complicated, as can be seen when looking at complex instruction caches [RQJS97] and value speculation mechanisms [GM96].

Given that, as technology evolves, an increasing number of transistors will be included on a single chip [Yu96], the question is how future processors will use these additional transistors. The majority of current processors have been focused on techniques aimed at exploiting more and more instruction level parallelism [Eme99] [Kah99]. However, measurements of actual performance of applications running on machines exploiting instruction level parallelism, show that the actual performance achieved falls very short

of the theoretical peak performance of the machines [CB96]. Many studies have pointed out that this lack of performance can be due to different effects, such as data and instruction cache misses, branch mispredictions, memory dependences or lack of program parallelism [Wal91] [LW92] [BGB98].

Therefore, although the exploitation of instruction level parallelism has yielded large performance improvements, and scaling current superscalar processors to achieve large amounts of ILP is an area of very active research, there is a growing consensus that this scaling can not be done by simply trying to fetch, decode and issue more and more instructions per cycle [PJS96] [PJS97] [AHKB00]. Some of the reasons are:

- First of all, an aggressive fetch and decode engine must be designed, which is far from being trivial due to branches as well as instruction cache bandwidth issues [PW94] [RBS96] [CMMP95] [RLPN+99].
- Second, an aggressive issue engine, with a large instruction window, is required to be able to feed a large number of functional units [HKLS00]. The instruction window lookup time increases quadratically with the window size [PJS97].
- Third, to issue a large number of instructions a heavily multiported register file is needed, which can both endanger the cycle time and consume a large amount of chip area [CGVT00].
- Also, sustaining multiple memory accesses per cycle requires a multiported TLB and cache, and their cost is also proportional to the number of independent memory ports [JNT97] [BGK96].

All these aspects make the scalability of superscalar processors expensive and strongly technology-dependent [AHKB00]. Moreover, even if these problems can be overcome with future technology, the performance results generally do not pay off the amount of chip area and the design effort required [LWS96] [QCEV99], as we will see along this thesis.

Therefore, we think that we should exploit more than one source of parallelism in order to overcome the scalability problems of current superscalar architectures. We

will first analyze, in the following section, the different sources of parallelism available in programs, and how they are being exploited in current processors.

1.2 SOURCES OF PARALLELISM

There are different sources of parallelism in programs: instruction level parallelism, thread level parallelism and data level parallelism. Let us analyze each of them in more detail.

Instruction level parallelism

Among the different sources of parallelism, the instruction level parallelism (ILP) has been one of the most exploited sources [RF93]. A program presents ILP whenever different instructions of a single control flow can be executed in parallel and the program result is not altered. The detection of the instructions that can be executed in parallel can be done at compilation time or at run time. In the first case, the compiler selects groups of instructions that can be executed in parallel [Gas89]. These instructions are typically packed in a single instruction. The architectures that exploit ILP in this way are called Very Long Instruction Word (VLIW) architectures because of the large multi-operation instructions that are generated by the compiler [KM89] [PSW91] [Gas91]. The Intel-HP's Itanium [Sha99] is an example of VLIW-based processor. It has built-in parallelism description to avoid "searching" for ILP that the compiler already knew about. The concept behind VLIW is to make hardware scheduling decisions visible to the compiler, which could make its own optimizations [Lam88]. VLIW processors also employ new techniques [EGK+94] [Rau93], such as predication and speculation, which allow a processor anticipating and performing calculations before the need for the calculation, or the validity of data, can be fully checked. Of course, the drawback of these approaches is that results of speculative executions are often tossed out when the data can not be validated, and the practice of profiling code for VLIW processors creates an unnecessary software burden.

The exploitation of ILP at run time is mainly supported in current superscalar processors [Joh91]. In these processors, instructions are fetched, decoded, renamed and sent to the execution queues where they execute when their operands are available. This way of detecting and exploiting ILP is more flexible as the hardware has full information about the dependences between instructions (as opposed to the limited static information available to a compiler) [RF]. Looking at current superscalar microprocessors roadmaps, there is a disparity on how to exploit ILP and improve performance, without incurring in excessive circuit complexity that may result in clock speed limitations [AHKB00]. On one end, the Alpha 21464 [Eme99] and Power4 [Kah99] processors are going for 8-wide issue. On the other, Intel's Pentium4 [Int00] is an attempt at extreme clock frequency with limited issue width. This processor has a trace cache [PW94] that delivers up to 3 micro operations per clock cycle to the core but, in return, it provides a very fast clock by using hyper pipelined technology.

Thread level parallelism

Another source of parallelism is the thread level parallelism (TLP). A program presents TLP if it can be decomposed in different threads, or groups of instructions, that can be concurrently executed, whether speculatively or not. This approach yields a system with higher throughput and better resource utilization.

One of the types of TLP is the simultaneous multithreading (SMT) [TEL95]. In this case, instructions from different threads coexist at the same time in the reorder buffer. In each processor cycle, the processor issues to execution instructions from different threads. A few bits of thread information in the instruction queues and per-thread rename tables are needed in this case. One example of this trend is the announcement of the Alpha 21464 [Eme99] being a simultaneous multithreaded processor. This processor can execute as many as eight instructions in a clock cycle. To fully exploit that potential, this processor uses out-of-order execution and special fetching techniques to create four virtual "thread processing units" that make the CPU look like a four-way multiprocessing system. This processor will be first used at the high-end server space, where multiprocessing-ready applications already exist. Outside that space, however, there is no such code, and to use the 21464's virtual thread processors optimally, ap-

plications would have to be tuned to the realities of the underlying hardware resources that are shared among the virtual thread processors. This effort in the software field is the major drawback of the thread level parallelism approach, as it requires a significant transition towards multithreaded software models.

There is also another approach for exploiting the TLP paradigm. It is called chip multiprocessing (CMP) and it is based on putting more than one processor core in a single chip. IBM uses CMP on its Power4 processor [Kah99], as well as Compaq does in Piranha [BGM+00]. This proposal does not suffer from the software support drawback, but incurs in the extra cost of putting two processors on a single large -and sometimes hot- die. The problem in CMP is the increasing number of pins that are needed, so the memory bandwidth becomes a bottleneck and prevents from achieving sustained performance.

Data level parallelism

Finally, the third another source of parallelism in programs is data level parallelism (DLP). A program presents data level parallelism whenever there is a piece of code (normally a loop) that executes the same operation over a stream of data [PH96] [Fly97]. These data are usually allocated in data structures like vectors or matrices. The operation can be carried out over consecutively allocated data, or over elements that are separated in memory a fixed amount of positions, called stride. The number of elements on which the operation is carried out is called the vector length. Exploiting data level parallelism in loops decreases the number of instructions and operations executed. Moreover, vector instructions provide with a large amount of work, as large as the vector length, that will keep the functional units busy for many cycles.

Although many processors have used DLP exploitation in their design [Rus78] [IW91] [Oed92], only two companies are currently manufacturing processors that exploit word level parallelism, NEC [vdSD01c] and Cray [BS00]. Over the last few years, however, there has been a growing interest on ISA extensions aimed at exploiting sub-word level parallelism, such as MMX [PW96], Altivec [NJ99], MAX [Lee96], VIS [Koh95] or MDMX [MIP97]. This is also a form of data level parallelism in which short data

are packed on a in single register and operations are carried out simultaneously on the different register elements.

All in all, some of the different parallelism paradigms discussed above are orthogonal from each other and in practice they are implemented together in a single processor. For example, the IBM Power4 [Kah99] has two processors on a chip, each of them issuing several instructions in each cycle. The Alpha 21464 [Eme99] executes up to four threads simultaneously, and each of them can issue out-of-order eight instructions per cycle. In the limit case, we could have several processors on a single chip, each of them could be simultaneous multithreaded and could extract ILP by issuing several instructions per cycle. This is an important idea that leads us to conclude that there are several natural sources of parallelism in programs, and each microprocessor tries to achieve high performance by exploiting some of them. In this thesis we rely on this idea, and we propose a processor design that exploits data-level parallelism coupled with traditional superscalar ILP execution. This processor will be backed with a especially designed cache hierarchy aimed at accessing vector and scalar data. Our design reaches performance values for numerical and multimedia applications that the superscalar processor can not achieve on its own.

Since our proposal is based on a fusion of ILP and DLP, let us take an in depth look at these two paradigms.

1.3 ILP PARADIGM

The ILP paradigm is mainly exploited in current superscalar architectures [Joh91] [SS95] [SFK97]. Table 1.1 presents the different characteristics of some of the current superscalar processors. These processors use different techniques aimed at detecting and exploiting instruction level parallelism:

- **Pipelining**. The pipelined execution of instructions exploits instruction level parallelism since different parts of the different instructions are executed simultaneously. In a pipelined datapath the execution of each instruction is carried out following

Processor	Alpha 21264C	AMD AthlonMP	HP PA-8600	IBM Power3-II	Intel Itanium	Intel Pentium4	MIPS R14000	Sun Ultra-II	Sun Ultra-III
Clock Rate	1001 MHz	1.2 GHz	552 MHZ	450 MHZ	800 MHz	1.8 GHz	500 MHz	480 MHz	900 MHz
Cache (I/D/L2)	64K/64K	64K/64K/256K	512K/1M	32K/64K	16K/16K/96K	12K/8K/256K	32K/32K	16K/16K	32K/64K
Issue Rate	4 issue	3 x86instr	4 issue	4 issue	6 issue	3 ROPs	4 issue	4 issue	4 issue
Pipeline Stages	7/9	9/11	7/9	7/8	10	22/24	6	6/9	14/15
Out of order	80 instr	72 ROPs	56 instr	32 instr	None	126 ROPs	48 instr	None	None
Rename Regs	48/41	36/36	56 total	16 int/24 fp	328 total	128 total	32/32	None	None
Memory B/W	2.66 GB/s	2.1 GB/s	1.54 GB/s	1.6 GB/s	2.1 GB/s	3.2 GB/S	539 MB/s	1.9 GB/S	4.8 GB/s
Die size (mm2)	115	128	477	163	300	217	204	126	210
Transistors	15.4mill	37.5mill	130mill	23mill	25mill	42mill	7.2mill	3.8mill	29mill
Availability	3Q01	2Q01	3Q00	4Q00	2Q01	3Q01	3Q01	3Q00	3Q01

Table 1.1 Main statistics for the key high-end processors available. Figure reproduced from [Mic01].

a set of sequential stages [Kog81] [PH96] [Fly97]. The number of stages depends on each processor [DF90], as can be seen in table 1.1. However, the main steps in the execution of an instruction are: fetch, decode, issue, execute and write results. In some processors these stages can be additionally divided into different stages, depending on the design decisions. In the instruction fetch stage, the instruction cache is accessed in order to read the instruction. The decode and rename stage decodes the instruction, renames logical registers into physical registers and predicts the outcome of the branch in order to be able to make a decision about the execution of the following instructions (wether sequentially continue execution or branch to the destination address). After that, the instruction is issued to the execution queues of the different functional units. Then, the instruction is executed in the appropriate functional unit (or accesses the data cache, if it is a memory access). The instruction finishes its execution writing the result in the register file, if it has to.

- **Multiple instruction issue**. The pipelined model previously commented can be evolved to deal with more than one instruction in each cycle [SV87]. In that case several instructions are fetched, renamed, issued, executed and finished in each cycle. Instructions executed in parallel can not depend on each other, of course. The early 1990s saw a number of processors, like DEC 21064 [McL93], HP PA

7100 [AAD+93] and MIPS R8000 [Hsu94]. All of them were implementations that achieved multiple issue by executing instructions of different types. In table 1.1 we can see that different current processor designs issue different number of instructions. The majority of them issue four instructions in each cycle, being Itanium [Sha99] the processor that issues the larger number of instructions (up to six instructions in each cycle).

- **Dynamic scheduling and Out-of-order execution**. The previous technique allows exploiting much more instruction level parallelism, but it also relies on the ability for finding several independent instructions that can be executed in parallel. This process can be carried out statically, at compilation time, or dynamically, at execution time. In the first case, the compiler selects groups of independent instructions and packs them in a single VLIW instruction [PSW91] [Gas91]. In the later case, instructions are fetched, decoded and issued in original program order, but once issued they can be executed out of program order, based on the availability of their operands. After execution, instructions are graduated in original program order in order to preserve the semantics of the program [DT92]. This technique provides the processor with additional flexibility in order to find independent instructions that can be executed, thus keeping the functional units busier.

All these techniques combined allow the computer to achieve high performance. Table 1.2 presents the best reported SPEC CPU2000 (base) results for each shipping vendor previously commented in table 1.1.

Of course, processors can reach even better performance if programs are compiled by knowing the underlying processor design. For example, the compiler can apply some compilation techniques, like loop unrolling or scalar replacement, in order to expose more parallelism between instructions, thus facilitating exploiting parallelism at execution time.

The two major challenges of an ILP processor are branches and cache memory misses. These two elements disrupt the instruction flow and prevent instructions from flowing through the pipeline. Cache misses are a problem because they stall the pipeline, thus decreasing performance. Several techniques have been developed in order to fight

Processor	Alpha 21264C	AMD AthlonMP	HP PA-8600	IBM Power3-II	Intel Itanium	Intel Pentium4	MIPS R14000	Sun Ultra-II	Sun Ultra-III
System or Motherboard	Alpha ES40 Model 6	AMD TyanThunder	HP9000 J6000	RS/6000 44P-170	Dell Prec. 730	Dell Prec. 330	SGI2200	Sun Enterprs 450	Sun Blade 1000
Clock Rate	1001 MHz	1.2 GHz	552 MHZ	450 MHZ	800 MHz	1.8 GHz	500 MHz	480 MHz	900 MHz
External Cache	8 MB	None	None	8 MB	2 MB	None	8 MB	8 MB	8 MB
164.gzip	466	608	376	230	332	663	266	165	348
175.vpr	482	319	421	285	262	330	460	212	384
176.gcc	636	357	577	350	359	649	367	232	491
181.mcf	374	229	384	498	187	506	607	356	474
186.crafty	803	665	472	304	355	586	409	175	442
197.parser	405	436	361	171	246	541	342	211	414
252.eon	798	836	395	280	414	795	433	209	463
253.perlbmk	605	759	406	215	309	828	305	247	456
254.gap	309	581	229	256	269	823	242	171	304
255.vortex	752	764	764	312	505	832	569	304	575
256.bzip2	606	401	349	258	286	471	404	237	503
300.twolf	821	417	479	414	356	444	552	243	481
SPECint_base2000	561	495	417	286	314	599	397	225	439
168.wupside	482	669	340	360	591*	897	321	284	413
171.swim	898	785	761	279	1369*	1294	310	285	319
172.mgrid	343	464	462	319	749*	639	249	226	233
173.applu	411	437	563	327	1022*	739	261	150	218
177.mesa	785	662	300	330	329*	671	348	273	466
178.galgel	1577	314	569	429	1019*	589	1154	735	906
179.art	1987	345	419	969	2369*	530	1213	920	973
183.equake	179	355	347	560	834*	816	229	149	210
187.facerec	996	417	258	257	637*	524	475	459	643
188.ammp	473	347	376	326	511*	399	475	313	409
189.lucas	511	480	370	284	837*	866	272	205	206
191.fma3d	415	446	302	340	323*	445	211	207	297
200.sixtrack	404	304	286	234	575*	307	248	159	282
301.apsi	506	315	523	349	350*	474	285	189	328
SPECfp_base2000	585	433	400	356	703*	615	362	274	369

Table 1.2 Best reported SPEC CPU2000 (base) results for each shipping vendor. Figure reproduced from [Mic01]. (*) Dell PowerEdge 7150 with 4MB L3 cache.

against the effect of cache misses [CB92] [SV97] [SC97] [Zha96] [Vei97] [RBS96]. However, there is always a certain number of compulsory cache misses that can not be eliminated.

Regarding branches, the problem are misspredictions. Accurately predicting branch outcomes is an important research topic [McF93] [YMP93] [SJSM96] [Zha96] [JSN98] on which there is a growing interest in the research community. Each time a branch is misspredicted, and the wrong path is being executed, the processor pipeline must be flushed and refilled with the appropriate instructions. In those situations there is a performance loss because of those cycles spent in executing the wrong instructions.

1.4 DLP PARADIGM: ANOTHER SOURCE OF PARALLELISM

The data level parallelism (DLP) paradigm uses vectorization techniques to discover data level parallelism in a sequentially specified program and expresses this parallelism using vector instructions [Rus78] [Oed92] [CGMW88] [NKT+95] [WKI86] [AJ88] [DH] [SK86] [Smi91] [CTS96]. A single vector instruction specifies a series of operations to be performed on a stream of data. Each operation performed on each individual element is independent of all others and, therefore, a vector instruction is easily pipelineable and highly parallel [Arn83] [HT72] [Ric78] [GBH96] [NKT+95].

There are two very important advantages in using vector instructions to express data-level parallelism. First, the total number of instructions that have to be executed to complete a program is reduced because each vector instruction has more semantic content that the corresponding scalar instructions. Second, the fact that the individual operations in a single vector instruction are independent allows a more efficient execution: once a vector instruction is issued to a functional unit, it will use it with useful work for many cycles. During those cycles, the processor can look for other vector instructions to be launched to the same or other functional units. It is very likely that, by the time a vector instruction completes all its work, there is already another vector instruction ready to occupy the functional unit. Meanwhile, in a scalar processor, when an instruction is launched to a functional unit, another instruction is required at the very next cycle to keep the functional unit busy. Unfortunately, many hazards can get in the way of this requirement: true data dependencies, cache misses, branch misspredictions, etc.

The combination of these two effects has many related advantages:

- First, the pressure on the fetch unit is greatly reduced. By specifying many operations with a single instruction, the total number of different instructions that have to be fetched is reduced. Many branches disappear embedded in the semantics of vector instructions [QEV98b].

- A second advantage is the simplicity of the control unit. With relatively few control effort, a vector architecture can control the execution of many different functional units, since most of them work in parallel in a fully synchronous way.
- A third advantage is related to the way the memory system is accessed: a single vector instruction can exactly specify a long sequence of memory addresses. Consequently, the hardware has considerable advance knowledge regarding memory references and can schedule these accesses in an efficient way [VLL+92] [PVAL95] [CEV98]. Moreover, in the DLP style of accessing memory every single data item requested by the processor is actually needed. There is no implicit prefetching due to cache lines. Additionally, the information on the pattern used to access memory is conveyed to the hardware through the stride information and it can be used to improve memory system performance [VLL+92] [VLPA95] [PVAL95].
- In addition, a vector memory instruction is able to amortize startup latencies over a potentially long stream of vector elements. Since each vector instruction works on a long stream of operations, functional unit latencies and memory latencies can be amortized across all vector elements. In the particular case of memory accesses, once a memory load operation is started, it pays for some initial latency, but then, assuming no memory conflicts, it can deliver one word per machine cycle.
- Finally, the DLP model can be easily scaled up to higher levels of parallelism by replicating the number of functional units and adding wider paths from the vector registers to the functional units. All this without increasing a single bit the complexity or the pressure on the decode unit. The semantic contents of the vector instructions already include the notion of parallel operations. This increase can be as large as the vector length.

All these advantages, which will be largely addressed in section 3.2, lead us to consider that it is worthwhile including DLP in future microprocessor trends. Moreover, including DLP techniques does not prevent from also exploiting ILP by issuing several instructions to execute. Our proposal is focused on the fine architecture level and consists in merging ILP and DLP techniques in a single chip processor. We strongly believe that adding DLP to an ILP processor can guide us to higher levels of paral-

lelism. Although our proposal does not include it, additionally, TLP techniques can also be applied at the coarser level to reach even higher performance levels.

Thesis goal

The goal of this thesis is to show that ILP and DLP can be merged in a single architecture to execute regular vectorizable code at a performance level that can not be achieved using either paradigm on its own. We will try to show that the combination of the two techniques yields very high performance at a low cost and a low complexity: the resulting architecture has a relatively simple control unit, tolerates very well memory latency and can be easily partitioned into regular blocks to overcome the wire delay problem of future VLSI implementations. Also, the control simplicity and the implementation regularity both help in achieving very short cycle times. Moreover, we will show that this architecture can be scaled up very easily, while scaling up an ILP processor is very costly in terms of hardware (and, at some point, may even not be feasible). Even if one scales up a superscalar, we will show that their performance falls behind the performance of the machine exploiting both ILP and DLP.

In order to reach high performance it is a key point of this thesis to also propose a novel cache hierarchy, tuned for the exploitation of the ILP and DLP paradigms. Each source of parallelism exposes a different way of accessing memory data. Therefore, the exploitation of each source of parallelism requires an especially designed memory hierarchy. In our case, we propose a memory hierarchy tuned to accessing scalar and vector data. This memory hierarchy is based on the "vector cache", which is a cache aimed at accessing vector and scalar data. This cache will be able to provide high bandwidth to the vector register file, to allow this bandwidth to scale up as we scale the functional units, to minimize conflicts between vector and scalar data and to guarantee that the processor cycle time is not in jeopardy due to the inclusion of a high bandwidth port to the vector register file.

Machine	Year	Clock Freq. (MHz)	Mflops/CPU	No. CPUs	Main Memory Bw/CPU	Load latency(ns)
Cray-1	1976	80	160	1	640 MB/s	150
Cray-XMP	1982	105	210	2	2.5 GB/s	123
Cray-2	1982	243	486	4/8	1.9 GB/s	200
Cray-YMP	1989	167	334	8	4 GB/s	100
Cray-C90	1992	243	970	16	12 GB/s	95
Cray-J90	1995	100	200	32	1.6 GB/s	340
Cray-T90	1994	450	1800	32	21 GB/s	70/116
Cray-SV1ex	2001	500	2000	32	6.4 GB/s	NA

Table 1.3 Evolution of the Cray vector machines.

1.4.1 DLP: Past, Present and Future

Given all the advantages mentioned above, one could ask what was the fault in the previous implementations of the DLP paradigm, that made them be slowly outshaded by other forms of computing.

Past of DLP

The DLP paradigm has been exploited using vector instruction sets [Rus78] [IW91] [Oed92]. The first vector machines were supercomputers using memory-to-memory operation [HT72] [Wat72], but vector machines only became commercially successful with the addition of vector registers in the Cray-1 [Rus78]. Following the Cray-1, a number of vector machines have been designed and sold, from supercomputers with very high vector bandwidths [Oed92] [KIS+94] to more modest mini-supercomputers [Con92] [PM86].

The peak performance of vector supercomputers has been constantly improving from the original 160 Mflops per processor of the Cray-1 up to 10 Gflops per processor of the more recent NEC SX-5 [vdSD01c]. This improvement has been achieved both through better cycle times (from 12.5 ns in the Cray-1 down to 3.2 ns in the SX-5) and higher number of floating point operations initiated per cycle. As an example of these improvements, table 1.3 shows the evolution of the Cray vector processors from the initial Cray-1 until the current Cray SV1.

Some years ago, parallel vector processors (PVP) were slowly displaced from the marketplace by other forms of supercomputing. There were several reasons:

- First, vector computers are not as general purpose as scalar processors. Vector processors achieve their highest performance values during the execution of vectorizable programs. For non vectorizable programs their performance results are very low since they can not take advantage of the latency tolerant vector code. Therefore, the flat memory system, without caches or with a small scalar cache, becomes a bottleneck when almost every data is accessed in scalar mode. Meanwhile, scalar processors obtain higher performance values for these non vectorizable programs, since their memory hierarchy is aimed at accessing scalar data.

- Second, because of the high cost of traditional vector processors. In order to keep the powerful cpu fed with enough data two related problems had to be solved. It was necessary to provide enough bandwidth from the memory system to the processor to ensure that the maximum processing rate is achieved. Moreover, this bandwidth had to be provided without increasing too much the latency delays that each memory request had to pay. Vector processors relied on high-performance, highly-interleaved memory systems that could sustain a bandwidth rate in accordance with their computing capabilities. Through the use of many parallel banks of interleaved memory (between 256 and 1024 banks depending on processor and configuration [SWL+91]) the necessary bandwidth was usually achieved. To keep the latency reasonably low, the fastest memory technology must be used. Vector processors tended to have SRAM/SSRAM memory modules [Pri96], which had a very high cost and, usually, turned out to be a very large fraction of the overall system cost. In comparison, their market adversaries leveraged commodity technology, which yielded a very low cost and delivered a very good price/performance ratios.

- Fourth, vector computers could not follow the impressive evolution of scalar processors in those years. Scalar processors using CMOS technology evolved very rapidly and their speed increased at a rate of 60% per year [PAY96]. Those scalar processors included a variety of architecture techniques, such as decoupling [Smi82], out-of-order execution [Tom67] [Yag96] and hierarchical memory systems based on

caches [KELS62] [Wil65] [Hil87] [Hil88] [Smi82] [SG83] [PH96] [LR91], that had greatly improved their processing power. This led to a situation where, for certain classes of cache-friendly problems, a microprocessor could match and sometimes exceed the performance of a vector supercomputer at a small fraction of its cost. While it was true that not all problems were well suited for cache-based machines, it is clear that those types of systems took a substantial share of applications that had previously belonged to the vector computing realm. As a result, vector processors were relegated to those classes of problems where the data size was too big to fit in a cache-based memory system.

- Fifth, the slow decline of vector processors can also be attributed to the fact that their architecture did not change much since the introduction of the Cray-1 [Rus78]. While superscalar microprocessors adopted many architectural features to increase performance while still retaining low cost, vector machines used almost the same architectural concepts introduced in 1976. In order to remain viable, and to extend to low cost systems, vector designs needed somehow evolve.

- Finally, despite the theoretical processing rates of vector cpus and the high bandwidth memory systems built around them, a constant in the evolution of vector supercomputers has been that the achieved performance of real programs has always been far from the theoretical peak performance [SH91] [Don93] [SH94] [Del91] [PB88]. What is worse is that this discrepancy between peak and actual performance occurred even for programs that were highly vectorized. As a result, the theoretical advantage of the vector computing paradigm over other forms of supercomputing was lost.

These most notable forms of computing that outshaded vector processors were large (MPP) and medium (SMP) scale multiprocessors built out of commodity microprocessors. The success of these other technologies were mostly based on leveraging CMOS microprocessor technology and DRAM main memory systems, yielding systems that had a high performance at a low cost. Meanwhile, PVP used expensive ECL technology, as in those days gate-switching speed was the limit on performance. This led to a shrinkage of the traditional PVP market niche. As a result, many companies devoted to manufacturing vector computers, like IBM, HP/Convex, Meiko, Alliant, Thinking

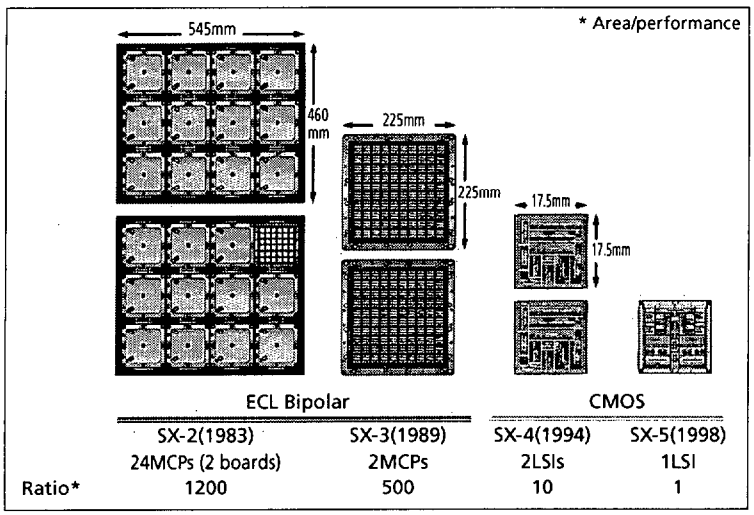


Figure 1.1 Vector processor migration from ECL to CMOS technology. Figure reproduced from [Lan98].

Machine and others, left the vector supercomputing business, and nowadays only two companies keep on manufacturing vector supercomputers: NEC and Cray.

Present of DLP

Over the last ten years these vector supercomputer manufacturers have made an important effort in adapting vector processors to the new technologies. All of them have moved from the costly ECL technology to CMOS technology, which lowers the fabrication costs and the power consumption appreciably. In the NEC case, the area/performance ratio has been reduced from 1200 down to 1 because of this evolution [Lan98], as can be observed in figure 1.1.

In the memory field, they have also migrated from expensive SRAM to commodity SDRAM chips, which provide a higher memory density, lower cost and higher performance. Figure 1.2 shows that in fifteen years the volume/memory capacity ratio has decreased from 500 down to 1, for NEC vector computers [Lan98].

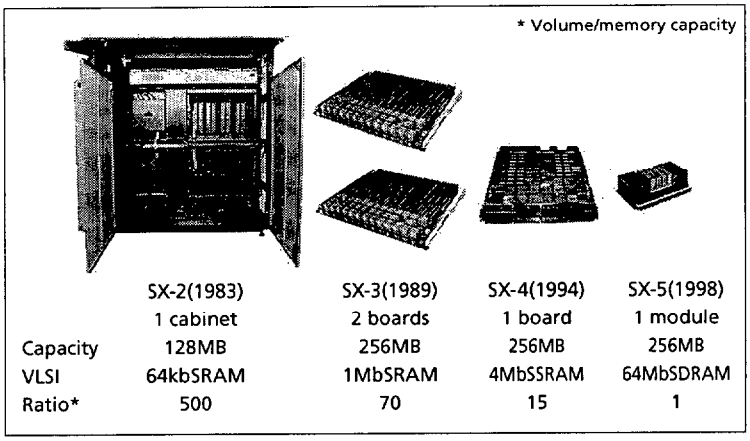


Figure 1.2 Vector processor migration from SRAM to SDRAM technology. Figure reproduced from [Lan98].

Nowadays, parallel vector machine vendors offer supercomputing to the user at a much lower cost and with a higher performance. NEC is currently selling the SX-5 [IKFN98] series. It was first introduced in 1998 and it is characterized by NEC as a machine that works well for vectors that are not very long and for programs that also contain a non-negligible amount of scalar code. This can be attributed to the lower multiplicity of the vector pipes in the vector units. Each NEC SX-5 processor works at 313 MHZ, that is 3.2 ns cycle time, and it can achieve a theoretical peak performance of 10 Gflops [vdSD01c].

Fujitsu has introduced in 1999 the VPP5000 series [Fuj01], which is the distributed-memory vector multiprocessor that succeed the former VPP700 systems [Uch97]. The main enhancements have been the reduction in cycle time, down to 3.3 ns, and the floating-point vector pipes are able to deliver floating multiply-add results. Each vector processor is able to reach a peak performance of 9.6 Gflops. Although the system was announced in November 1999, the first performance results have only recently been available [vdSD01b].

Finally, Cray has recently introduced the new SV1[BS00], a shared-memory multi-vector processor. The Cray SV1 is built using CMOS technology and it is the successor of both, the CMOS-based Cray J90 and the Cray T90, which was based on ECL

technology. Although the initial cycle time was higher, they have announced a 2 ns cycle time by the middle of 2001 [BS00], and each processor will reach a peak performance of 2 Gflops. The very new characteristic of the SV1 vector processor is the addition of a 256 Kbyte cache, as we already proposed at [QCEV99]. In the same way as we did, the cache is combined with the cpu onto one ASIC. Their initial step, however, is simpler than our proposal. Although no independent measured performance is available at the moment, some tests have shown the importance of the cache in SV1 [vdSD01a]. For large sized vectorizable problems the presence of the cache has almost no effect. However, for modestly sized problems the cache can boost the performance with a factor of 1.5 up to 2. The reason is probably that with large sized problems the program's working set does not fit in cache, so program execution does not benefit from exploiting temporal locality. Therefore, many memory accesses miss in cache and data must be brought from the main memory at a 6.4 GB/s bandwidth. On the contrary, for modestly sized problems the program's working set fits in the cache and data can be reused, thus benefiting from the 14.4 GB/s processor-cache bandwidth.

Besides, the DLP paradigm has been also recently introduced in commodity microprocessors. During the last years, PC and workstation users unknowingly have been using a restricted form of DLP, since the commodity chip makers have began incorporating SIMD hardware into their mainstream superscalar processors. However, it is important to distinguish our proposal from this tendency in microprocessor design targeted at the exploitation of sub-word parallelism. Most major computer vendors have recently included multimedia specific instructions in their architectures such as MMX [PW96], VIS [TONH96], MAX [Lee96], MDMX [MIP97] or Altivec [NJ99]. Except for the Altivec case, all other extensions only offer sub-word parallelism. That is, a 64-bit register can be broken into independent entities of 8, 16 or 32 bits that are operated on in parallel. Although the term "SIMD" is used to refer to these instructions, they are a restricted form of vector computing, and care must be taken to distinguish traditional vector-like SIMD operation and sub-word SIMD operation.

In this thesis we do not focus on sub-word level parallelism. Rather, we integrate a full vector unit in an out-of-order superscalar processor. Our proposal has true vector instructions where a single instruction operates on multiple, independent 64-bit words

(a vector register). The advantage of having full-blown vector units is that numerical applications can benefit greatly from them while, typically, they do not take advantage of MMX-like ISAs. Furthermore, some multimedia applications that are not amenable to exploit sub-word parallelism can also take advantage of our vector units. Of course, although beyond the scope of this thesis, nothing would prevent our vector units to also include sub-word parallel instructions, such as the ones provided by Altivec.

Future of DLP

Future trends in computing are very difficult to predict. What is clear, however, is that those trends will include exploiting DLP. At the supercomputing high-end, NEC and Cray will still keep on manufacturing vector processors. In fact, NEC has recently announced that there will be at least three more generations of the SX processor, called SX-6, SX-7 and SX-8 [NEC01], which give us an idea about future expectations about vector supercomputing for this company. Meanwhile, the Cray SV2 system, currently in development, will be the next supercomputer from Cray [Inc01].

At a lower-end, it is expected that the interest on SIMD extensions for superscalar processors will increase, in order to meet the increasing users demand on media processing [Wil98], such as video and audio processing, image rendering, and other media applications [DD97] [LW97] [BE98]. Also at that end, as technology continues to evolve, miniaturization will make the space needed for a single processor much smaller than the actual chip size [Lan00]. As a processor is 0.1 square cm in size, and the chip is 6.2 square cm, there is room for multiple processors [Kah99]. However, that is not the only way. The alternative, which is proposed in this thesis, is to complement the superscalar processor with vector processing and large, especially designed cache memory hierarchies.

1.5 RELATED WORK

Over the last few years, there has been an increasing interest on studies regarding DLP. These studies deal with the different ways of exploiting DLP: vector processors and sub-word level parallelism.

In the work presented in [LD97] and [LS98], it is proposed the use of simple vector processors in future desktop systems. Although this may sound similar to our proposal, indeed the differences are profound. Lee argues that by using vector units, one can keep the scalar core in-order (dispensing with the expensive and slow out-of-order features) and thus favor high clock rates. Although we agree with the argument, we claim that any feature that goes against integer performance will not be adopted by chip vendors. That is, to successfully add a vector unit to general purpose microprocessors, the vector unit has to be perceived as an add-on that does not disrupt, slowdown or interfere with the performance of the integer core. Therefore, if the current tendency is to include out-of-order execution even in high frequency designs such as the Alpha lineage, the vector unit must be adapted to out-of-order execution and register renaming.

Furthermore, Lee does not discuss the implications of the vector unit on the memory system, while this thesis is mainly devoted to designing a feasible cache hierarchy that fits both the scalar engine and the vector unit.

In [EVS97] [Esp97] Espasa makes the case for an out-of-order vector processor. The study starts from an obsolete in-order vector architecture [Con92]. Although this processor issued only one instruction in each cycle there was overlapping between scalar and vector instructions. However, a stall in a vector instruction prevented further vector instruction dispatching until the hazard was resolved. It was a register-to-register machine with few vector registers, and connected to the functional units through a crossbar. The main memory system was a flat multi-banked one, connected to the processor through a single memory port. The study is focused on applying register renaming and out-of-order execution to this vector processor, while keeping its 1-way nature, in order to improve performance. In [EVS98] they identify two possible evolutions of vector architectures. One of them, that they studied in [EV97], is the proposal

of a simultaneous multithreaded vector processor able to be implemented with one billion transistors. This proposal is aimed at high end vector supercomputers. The other possible evolution, situated at the mid-range servers, is what they called the "Micro Vector", which uses out-of-order execution, short vector registers and specialized caches.

Our proposal, which is more concerned with the "Micro Vector" that they define, does not deal with vector processors at all, we start from a full superscalar architecture and we study what is the minimum additional components that must be included in order to introduce vector computing inside. We also propose a new cache hierarchy, based on a traditional one, aimed at accessing vector and scalar data.

Villa [VEV97] [VEV98] studies the importance of the vector registers length in vector processors. For different vector lengths, it analyzes performance and cost. Our study will include results for 16-element and 128-element vector registers, as we consider this exploration essential in determining the minimum vector elements required to achieve a performance improvement. The fact that the memory system that Villa uses is flat and multi-banked, while our proposal is a cache hierarchy, make their behavior different. Therefore, its conclusions can not be directly applied in our field.

In the sub-word level DLP exploitation research field there have been different approaches. Although these studies deal with DLP exploitation, they only exploit sub-word level parallelism while we focus on word level parallelism exploitation. Moreover, they only study multimedia applications, while we are also interested in numerical applications. Finally, some of these studies are restricted to embedded processors, while ours are aimed at general purpose processors. As discussed above, our work could be additionally improved with the inclusion of sub-word level instructions. Therefore, these works complements, rather than overlaps, ours.

Among these studies, [CEV99] proposes the Matrix Oriented Multimedia (MOM) ISA extension by fusing conventional vector ISA approaches together with media ISA extensions like MMX [PW96]. MOM instructions can be viewed as vector versions of sub-word parallel instructions, that is, they operate on matrices where each row corresponds to a packed data type. MOM is targeted at small matrix structures typically

found in multimedia applications. This proposal has been additionally extended in [CEV01]. In this work the MOM ISA extension is combined with packed accumulators (as found in MDMX [MIP97]) in order to deal with reduction operations. These extensions outperform MDMX and MMX extensions.

In [SCEV99] an evaluation of different DLP oriented embedded architectures for a media workload is presented. This study includes SIMD ISA extensions, a traditional vector ISA (with long and short vector registers) and MOM ISA extension. For the embedded domain, the SIMD-like architecture arises as the more cost-effective option.

In [JTVW01] the Complex Streamed Instruction (CSI) set is presented. It is an ISA extension with some different characteristics to previous ones: CSI instructions process two-dimensional data streams (as MOM does), there is no architectural constraint on the length of the data streams, and it does not include pack/unpack instructions as conversions between different packed data are performed internally in hardware.

Another related proposal is the Imagine media processor [Wil98] [RDK+98] which has a load/store architecture for one-dimensional streams of data records. Imagine is organized around a large 128 KB stream register file, and consists of 48 functional units grouped in 8 arithmetic clusters. All operations are performed by transferring streams to and from the stream register file. Memory instructions transfer streams between the stream register file and memory. Stream computations are performed by passing a stream from the stream register file through the arithmetic units and back to the stream register file. In this processor the individual stream elements may be operated on in parallel to exploit data level parallelism. Instruction level parallelism can be exploited within the individual computation kernels. Imagine is suited for applications performing many arithmetic operations on each element of a long, one-dimensional stream. It seems less suited when only a few operations on each record are performed or when the vector length is small.

Another research topic that we deal with in this thesis, is the use of specialized cache hierarchies for ILP+DLP processors. In this sense, it is interesting to comment the cache-based memory system recently appeared in the Cray SV1 [BS00]. As we already

proposed in [QCEV99], this processor has a 256 Kbyte cache that is combined with the cpu onto one ASIC. Their proposal, however, is simpler than ours in several aspects:

- First, their cache is unified for instructions and scalar and vector data. Our proposal, however, follows the traditional cache designs of separated instruction and data caches [QCEV99] [QCEV01].
- Second, they propose a unique first level cache. Meanwhile, we propose a two-level cache hierarchy system [QCEV99] [QCEV01], as we will explain in detail in chapter 4. We will study two cache hierarchy models. In the first one both, scalar and vector data, are located in both cache levels and they behave similarly to a traditional cache hierarchy. In the second model the first level cache only contains scalar data. Vector data is located in the second cache level and a direct path from the processor to that cache allows accessing them [Hsu94] [Sha99]. Our proposal goes further away exploring the separation of vector and scalar workload.
- Third, the cache line size in the SV1 cache is just one word. This is by no mean common in a cache, which has traditionally been used to exploit both, temporal and spatial locality. This cache line size prevents from exploiting spatial locality, which is fully present in the instruction flow and in stride-1 vector accesses. The only reason to set such small cache lines is to avoid moving useless data when non stride-1 or gather/scatter vector memory accesses are executed. When multiple-word line sizes are used, these types of accesses spend some memory bandwidth as some data are moved to the cache, but they are not actually used later. However, there are two approximations in order to solve this problem: the first one, adopted in SV1, is to sidestep the problem by using 1-word cache lines, thus losing the opportunity of improving performance by exploiting spatial locality. The second one, which we propose [QCEV99] [QCEV01], consists in facing the problem and designing a memory hierarchy that deals well with strided memory accesses, as we will see in chapter 6.
- Closely related to the previous consideration is the fact that the SV1 cache delivers four data elements to the processor in each cycle [vdSD01a]. As the cache line size is just one word large, four independent memory accesses have to be done to four different cache lines. Therefore, four independent memory ports are needed,

which is an expensive solution. In our proposal, however, the multiple-word line size allows having just one four-word width memory port [QCEV99] [QCEV01]. A vector memory access goes to one cache line and brings the four consecutive elements, which is a lower cost solution.

1.6 A SUITABLE APPLICATION SPACE FOR EXPLOITING DLP

As stated in section 1.4, page 11, the DLP paradigm is expressed by means of vector instructions. However, not all tasks can be expressed by using vector instructions, and even inside a vectorizable program some parts will not be amenable to be formulated in a vector form. Therefore, from the DLP point of view, we can say that a program has two types of regions, data-parallel regions (D-regions) containing those zones that can be expressed with vector instructions, and scalar regions (S-regions), whose contents can not be formulated with vector instructions.

The DLP paradigm can be used to improve the performance of D-regions, once expressed with vector instructions. In general, the larger the fraction of the program inside D-regions, the better the performance improvement that can be achieved. The degree of vectorization of programs is a key parameter that determines the maximum speedup achievable through DLP processing. In fact, Amdahl's law [Amd67] [PH96] shows that a balanced vector and scalar performance is also necessary to achieve high performance. Therefore, the analysis inside S- and D-regions is essential when trying to increase programs performance using DLP. The ability to identify and separately study the behavior of a program inside D-regions and S-regions will allow us to predict and understand its performance behavior. For that reason, another novel contribution of this thesis is the identification, separation and study of the S- and D-regions of each program, both at the performance and instruction set architecture level.

Among the different types of programs amenable to be vectorized, it can be said that, in general, DLP can provide large speedups mostly for highly regular, vectorizable, applications. This has been the trend at the high-end vector supercomputers where tradi-

tional applications in the fields of environment and climate modeling [LL00] [Mar98b], computational fluid dynamics [Iko00] [Rhy99], reactive flows [TLO98] [FLTB99], astrophysics and geophysics [Mar98a], electromagnetism and plasma physics [Kat00], applied mathematics and theoretical physics [Tsu99], complex molecular systems and biology [Kan97] [Him00], quantum chemistry and material sciences [Ran96], among others [NEC98] [Him96], where only tractable by using this kind of machines. However, the increasing performance of each individual low-end microprocessor, and the possibility of interconnecting these processors to achieve large sustained performance is facilitating the movement of those traditional vector applications to the low-end supercomputers. For that reason, our proposal of an ILP+DLP processor will deal especially well with those kinds of applications.

Moreover, as general purpose microprocessors have continued to become more powerful, they have been asked to perform increasingly complex tasks. In fact, the increasing performance of general purpose microprocessors has not met the requirements of the networking and telecommunications infrastructure industry due to several emerging applications and trends. Example applications include the explosive growth of the Internet, the emergence of new digital communications technologies, including digital cellular phones, IP-based telephony, fax and multimedia, and wireless messaging [DD97] [LW97] [BE98]. A general trend in the industry is using programmable processors to implement adaptative filters, modulators/demodulators, and other functions previously only possible in hardware. These demanding new applications, along with the continually increasing needs of the computing market, can be met with our proposal of superscalar+vector architecture. We believe that, in the future, our proposal will be a single chip solution to the highest level of processing performance while expanding the processor's capabilities to concurrently address high-bandwidth data processing and the algorithmic intensive computations which are typically handled off-chip by other devices, such as dedicated hardware, DSPs or custom ASICs.

Accordingly, our experiments on the ILP+DLP architecture will be carried out by using these types of applications as our benchmark set. We have selected programs from both, the numerical and multimedia fields in order to study the performance answer of our proposal. Our set of benchmarks covers traditional vectorizable floating

point applications plus the ever-growing number of computationally and bandwidth intensive media tasks.

Finally, we would also like to mention a key topic in including vector instructions at the instruction set architecture level: the compiler. In order to make use of a vector functional unit, compilers must include vector compilation techniques that can expose data level parallelism, so that programs can be vectorized. Therefore, compiler developers will have to make an effort to include these techniques, which in fact they are currently making in order to take advantage of the sub-word level parallelism exploitation. However, they can benefit from the large experience in vectorizing compilers at the vector supercomputers level [AJ88] [Bos88] [Bra86] [Bud84] [DH] [SK86] [Smi91] [CTS96].

1.7 THESIS OVERVIEW

In this thesis we show that ILP and DLP can be merged in a single architecture to execute numerical and multimedia applications at a performance level that can not be achieved using either paradigm on its own. We propose a current generation superscalar processor enhanced with a vector unit. The proposed architecture is backed with a new cache hierarchy that includes a vector cache.

The first topic we deal with is a detailed analysis of the instruction level characteristics of a set of numerical and multimedia applications. This study proves that vector programs execute fewer basic blocks, instructions and operations than superscalar programs. Therefore, less aggressive fetch and decode units are needed, which can yield a faster clocking of the datapath.

The analysis inside S-regions and D-regions shows that superscalar programs execute many more basic blocks, instructions and operations inside D-regions than the vector programs. The analysis inside S-regions shows that the quality of the scalar code generated by the superscalar compiler is higher. The solution to improve the quality of the vector programs code consists in building a new set of hybrid benchmark pro-

grams starting from the pure superscalar and vector programs. Each hybrid vector program consists in the original S-regions from the scalar version plus the original D-regions from the vector version. The hybrid vector programs execute fewer instructions and operations than the pure vector programs while keeping almost the same vector characteristics.

From these studies we state that, given the benefits of the vector ISA, it is worth exploring the possibility of including a vector functional unit in a current superscalar architecture. Therefore, we outline the design of an ILP+DLP architecture.

The next topic presents the performance results of our proposed superscalar architecture with a vector unit (SSV), compared with a traditional superscalar (SS) processor. On one hand, the study of scalability and potential performance of the SSV and SS architectures with a perfect memory shows that the SSV architecture scales very well as more memory and computing resources are added to the processor. Moreover, it reaches higher values of parallelism than the SS architecture with a lower cost and control complexity.

The analysis inside D-regions and S-regions also shows that the SSV architecture achieves high values of parallelism inside D-regions. However, their contribution to the overall performance is determined by the relative weight of D-regions in the whole programs. As it is expected, the performance inside S-regions is better for the SS architecture. The reason is that the superscalar core of the SSV architecture remains constant along the different configurations, while, in the SS architecture, the core is scaled adding more and more resources.

On the other hand, we also study the performance of the SSV proposal when a real cache hierarchy is introduced. We study two different cache hierarchies, based on the introduction of the vector cache. The two models differ on where the vector cache is located, in the first or second cache level. As a first step, we study the cache hierarchy efficiency of both models. This study shows that numerical programs make a higher pressure on the main memory, and that the model with the vector cache in the second

cache level reacts much better to this pressure with lower main memory traffic and higher hit rates.

The study of the vector cache stall time reveals that this cache can spend a large percentage of the total execution time stalled due to different reasons. The two main reasons are full MSHR and WB entries. The other two reasons (coherence and cache conflicts) contribute in a lower degree to that bottleneck.

The performance evaluation with a real memory shows again that numerical programs are limited by the memory system, while multimedia programs are not. The study inside regions presents that, for the SSV architecture, programs scale well inside D-regions and behave constant inside S-regions.

As a general result, we conclude that the SSV architecture is a feasible architecture from the performance point of view. It reaches better performance than a traditional SS architecture, either with ideal or real memory systems, as the processor configuration is scaled. It is a good choice for multimedia programs and it achieves really good results for numerical programs.

However, these results can still be improved by tuning the memory hierarchy. Increasing the memory non-blocking mechanisms in the SSV architecture diminishes the negative effect of the vector cache stall due to the filling up of the MSHR and WB structures. The performance improvements obtained with this tuning reaches up to 37% for 128-element vector registers. Adding an extra port for scalar accesses is a profitable enhancement for running an heterogeneous set of programs, that can even include low vectorization programs, in the SSV, as it provides with some additional flexibility in accessing memory. Increasing the main memory bandwidth in the SSV architecture provides with increasing performance in those programs with large memory traffic.

We also study the effect in performance of future evolutions in microprocessor integration, which will motivate changes in the cache hierarchy design, including increasing its size, associativity and latency, while the memory bandwidth is also increased. Programs that are constrained by the memory bandwidth improve their performance. Meanwhile,

the rest of the programs suffers a performance loss because of the prevail of the higher latency over the reduced main memory traffic.

Finally we study two additional cache hierarchies aimed at dealing with the strided vector memory accesses. Results show that these new designs, although more expensive, can deliver improvements over the base case that range between 3% and 18%. Although these cache designs are complex and costly to implement, the large performance improvements obtained pay off the effort, so that as technology evolves they can actually be implemented.

1.7.1 Structure of this work

This work is organized as follows:

- Chapter 2 presents our working environment, including our tracing and simulation tools, the set of benchmarks, as well as the performance metric that we will use in the following chapters.
- Chapter 3 presents a detailed instruction level characterization of the benchmark programs, including distribution of instructions, operations, data types, vector characterization and analysis by regions. We finally introduce the hybrid version of the benchmarks used in the following chapters.
- Chapter 4 describes the superscalar+vector architecture (SSV architecture) that we propose. After describing the general datapath we focus on the new cache hierarchy proposed in this thesis, which includes the introduction of the new vector cache.
- Chapter 5 makes a scalability study for both, the traditional superscalar and the superscalar+vector architectures. The study is carried out, first, with an ideal memory system, and second with a real cache hierarchy. In the latter case we also make a cache efficiency study, by exploring hit/miss rate and traffic with the main memory, and a vector cache stall time time. These studies identify the memory bottlenecks that prevent the SSV architecture from attaining even higher performance.

- Chapter 6 performs some changes in the proposed memory hierarchy aimed at improving performance. Several changes to the architecture presented are proposed: enhancing the memory non-blocking mechanisms, improving memory bandwidth and adding a memory port for scalar accesses. We also study the effect in performance of future microprocessor integration. Finally, we introduce two additional memory systems aimed at attacking the problem of strided vector memory accesses.
- Finally chapter 7 summarizes the contributions of this thesis and presents open areas for future research.

2

TRACING AND SIMULATION TOOLS, BENCHMARKS AND METRICS

Summary

This chapter presents the different tools that have been used for carrying out this thesis. We describe the scalar and vector tracing tools, which have been used for extracting scalar and vector program traces, respectively. We also describe the simulation tools necessary to study the performance characteristics of programs running on an ILP+DLP architecture. We briefly present the vector and scalar machines from which traces have been extracted, including the functionality of both machines and the software tools necessary to gather traces of programs running on them. We introduce our benchmarks and discuss the necessity of carrying out some modifications in order to study different vector lengths as well as the benchmarks behavior inside regions. We analyze the quality of the scalar code generated by the vector compiler and we propose a way of improving it by creating hybrid benchmarks. Finally, we define the performance measure that will be used in the following chapters.

2.1 INTRODUCTION

This study carries out the analysis of an ILP+DLP architecture executing numerical and multimedia applications. We have used trace-driven simulation as the base of our experimentation methodology. The proposal presented in this work has been evaluated by comparing the performance of the trace driven simulations of the ILP architecture against the performance of the ILP+DLP architecture.

2.2 TRACING AND SIMULATION TOOLS

As already mentioned, we have used simulation as the base of our experimentation methodology. The advantages of trace driven simulation are the ease of reproduction of results and the highly controlled environment in which we execute the experiments. Since the trace is fixed, the results of different simulation runs can not be altered or influenced by external factors, such as machine load or memory conflicts.

We will now briefly describe the software tools that we have used in our studies. The first tool, a tracing tool, has been targeted to a Convex C4 [Con93] and is able to gather static traces of binaries running in vector mode on a single processor of a Convex C4 computer. The second tool is also a tracing tool. It has been used to gather dynamic traces of scalar binaries running on an Alpha 21264 processor [Gie97] [Kes99]. The third tool is a general simulator, which is able to be parameterized as the ILP+DLP architecture under study. This simulator can also be configured to act as an ILP processor.

2.2.1 Vector tracing tool: Dixie-c4

Dixie-c4 is a pixie-like tool adapted from the original Dixie tool [EM94] in order to deal with Convex C4 [Con93] binaries, rather than Convex C34 [Con92] ones. The Convex C4 architecture has more vector registers, which eliminates the large amounts of spill code of the C34 code. It also has more vector facilities, like the vector first execution

and the addition of new vector instructions, and an improved compiler aimed at better vectorizing programs.

Dixie-c4 is able to produce a static trace of basic blocks executed as well as a trace of the values contained in the vector length (VL) register. The ability to trace the values of the vector length register is critical for accurate detailed simulation of the program execution, because each vector instruction can execute potentially with a different vector length. Thus, our measurements do not suffer from the problems reported in [VSH91] [RR94]. It is also important to note that Dixie-c4 instruments the output of a commercial compiler without requiring any special properties in the binary, and that this tracing method gives a high precision in all of our measurements. Traces gathered using Dixie-c4 are physically stored in different files in disk. Static traces have some advantages over dynamic traces. First, by having traces stored in disk we do not depend on the Convex C4 being operative in order to be able to execute simulations. We just gather the traces, move them to another machine and execute the simulator, which opens the different trace files and consumes the trace according to the way the binary program dictates. This is an important characteristic, since the Convex C4 machine is nowadays obsolete. This machine is no longer supported by HP, which did not provide any software patch for the so called "2000 effect". However, the drawback of static traces comes from their size. Current SPEC benchmarks [SPE] generate trace files that require several tens of gigabytes on a hard disk. Given that we need around ten programs, we found that we need more than seven hundred gigabytes, just for a version of the programs. As we will see later, our studies require to have different versions of each benchmark program, which means several traces for each program, thus demanding a huge amount of hard disk space. In our case, this factor was a key restriction in the election of the benchmark programs.

The tracing procedure is illustrated in figure 2.1: the benchmark programs are compiled on a Convex C4 machine using the Fortran or C compiler at optimization level -O2, which enables vectorization. Then, the executables are processed using Dixie-c4, which decomposes the executables into basic blocks and instruments the basic blocks to produce six types of traces: a basic block trace (BB trace), a trace of all values set into the vector length, vector stride, vector first and vector mask registers (VL trace, VS

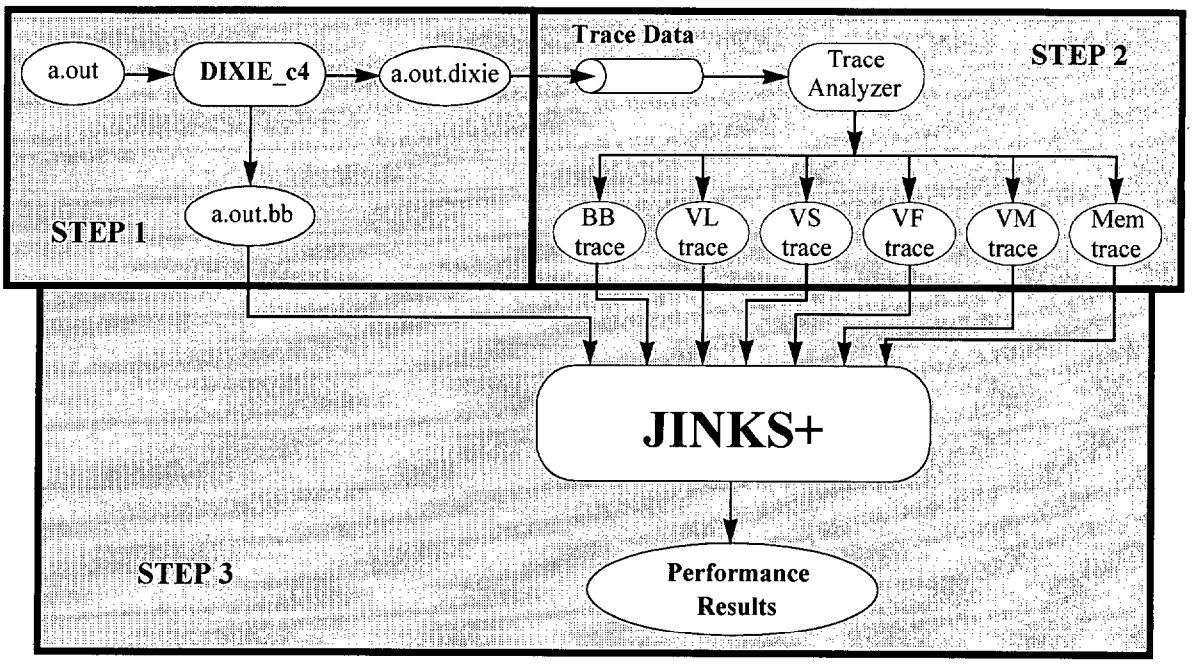


Figure 2.1 The instrumentation and simulation processes. Step (1) consists in processing a program's executable and generating an instrumented version of it. In step (2) we run the modified executable on the Convex C4 machine and we obtain a set of traces that fully describe the execution of the program. In step (3) this set of traces is fed into the simulator, which will do a cycle-by-cycle execution of the program and will gather performance results.

trace, VF trace and VM trace), and a trace of all memory references (actually, a trace of the base address of all memory references) (Mem trace). Dixie-c4 instruments all basic blocks in a program, including all library code. This is especially important since a number of fortran intrinsic routines, such as SIN, COS or EXP, are translated by the compiler into library calls. These library routines are highly vectorized and tuned to the underlying architecture and can represent a high fraction of all vector operations executed by the program. Thus, it is essential to capture their behavior in order to accurately model the execution time of the programs.

Once the executables have been processed by Dixie-c4, the modified executables are run on the Convex C4 machine. These runs produce the set of traces that accurately represent the execution of the programs.

2.2.2 Scalar tracing tool: Atom

Atom is the Compaq tracing tool [SE94]. It provides a diverse set of tools ranging from basic block counting up to cache modeling. It provides the common infrastructure in all code-instrumenting tools, leaving to the user the task of specifying the tool details. The user indicates the points in the application program to be instrumented, the procedure calls to be made, and the arguments to be passed. Starting from these specifications it generates dynamic traces, which means that the trace items are directly passed from the benchmark program to the simulator without using files on disk. While this characteristic eliminates the limitation of the trace length, it is indispensable to run both, the instrumented program and the simulator, on a Compaq computer.

2.2.3 Parameterizable Simulator: Jinks+

Jinks+ is a parameterizable simulator developed as an improvement of the previous Jinks vector simulator [Esp95]. It has been improved with a superscalar core which includes a reorder buffer, dynamic scheduling, out-of-order execution and branch prediction. A variety of accurate memory system simulation modules has been added, which allows simulating both, a memory system based on a multi-level cache hierarchy, and also a typical cache-less vector memory system.

The traces gathered using either Dixie-c4 or Atom are then fed to Jinks+ (see figure 2.1). Jinks+ closely models the behavior of a program execution on both, the ILP and ILP+DLP architectures. It does cycle-level simulation of the architectures and keeps detailed information about resource usage, number of hazards produced, stop issue conditions, etc. By simulating both architectures with the same tool we guarantee a comparison between them which is more fair than if different simulators were used.

2.3 CONVEX C4 VECTOR ARCHITECTURE

We have gathered vector traces from a Convex C4 machine [Con93]. The Convex C4 vector architecture is a register-register architecture. It consists of a scalar unit and a vector unit connected through a single memory port to an interleaved memory system.

The architecture defines three different sets of registers: A, S and V. Registers of type A, the address registers, are used to generate the base address of all memory references. Thus, all types of load/store instructions use at least one A-register in their addressing modes. There are 32 32-bit address registers (A0 through A31). The scalar registers, S registers, are 64 bits wide and are used to carry out all integer and floating point computations in scalar mode. Also, S registers are used in vector instructions whenever a vector is operated with a scalar value. There are 28 different S registers (S0 through S27). The vector registers, V registers, hold 128 64-bit words and are used for all types of vector computations, both integer and floating point. There are 16 vector registers (V0 through V15).

There are four different instruction formats. Two of them define 32-bit instructions, and the others define 64-bit instructions. All the instructions, except branches, have three operands. Two of them specify the source operands and the other specifies the destination. The source operands are usually registers, but one of them can be replaced by an immediate operand.

The scalar unit executes all instructions that involve scalar registers (A and S registers). There is a 32 KB data cache that only holds scalar data, which is kept coherent with the vector reference stream by serializing all memory references (scalar and vector) through a single port. Vector stores invalidate the necessary lines in the data cache so that subsequent scalar memory accesses are forced to reach main memory to get the most up to date value. Vector loads never read the scalar cache.

The vector unit consists of three computation units and shares the memory accessing unit with the scalar part. One of the units is a general purpose arithmetic unit capable of executing all vector instructions. The other two are restricted functional units that

execute all vector instructions except multiplication, division, square root, masking, merging, compress and expand instructions. All the functional units are fully pipelined.

Vector operations are performed under control of the vector length register. This 8-bit scalar register indicates the desired length, from 0 to 128 elements, of vector instructions. The vector first register also controls the execution of the vector instructions and indicates the first element of the vector register on which to execute the operation. The use of this register improves the reuse of the data in the vector registers. Vector memory operations use a third control register, the vector stride register. This register indicates the stride, in bytes, that separates the elements that must be fetched from memory. Finally, the vector mask register is a 128-bit wide register that allows all vector operations to be done under the control of a mask. Each bit in the vector mask register determines if individual operations of a vector instruction, already executed, are valid according to a condition, or not.

The sixteen vector registers are organized as four banks of four registers each. The connection to the functional units is carried out through a restricted crossbar, as each bank has three read ports and one write port. The compiler is responsible for scheduling vector instructions and allocating vector registers so that no port conflicts arise.

The Convex C4 that we have used for gathering traces works at 140 MHZ. It runs the ConvexOS operating system, Release V11.1. We have used the C and fortran compilers from Convex, as the set of benchmarks are written in both languages. We used the Convex fortran compiler, version 9.1, and the Convex C compiler, version 6.1. All codes have been compiled using the -O2 flag, which includes all scalar optimizations plus vectorization and vectorizing optimizations. The next optimization level implied parallelization and thus, was not used. All binaries were fully linked, that is, no shared libraries where used.

The quality of the vector code is good, in general, and the vectorizer performs relatively well on most vectorizable loops. The major limitations, as we will see later, come from the quality of the scalar code, as it is not as good quality as the code generated by the scalar compiler.

All these characteristics of the Convex C4 computer, and the in-order execution affect the instruction scheduling that the compiler carries out. However, the execution of the program traces is not handicapped thanks to the out-of-order execution of the simulated machines.

2.4 ALPHA EV6 SCALAR ARCHITECTURE

We have gathered scalar traces from an Alpha 21264 processor [Gie97] [Kes99]. Alpha is a 64-bit load/store RISC architecture, designed with particular emphasis on clock speed, multiple instruction issue, out-of-order execution and multiple processors.

Each Alpha 21264 processor has a set of registers that hold the current processor state. There are 32 integer registers (R0 through R31), each 64 bits wide. R31 reads as zero, and writes to R31 are ignored. When R31 is specified as a register source operand, a zero-valued operand is supplied. There are 32 floating-point registers (F0 through F31), each 64 bits wide. When F31 is specified as a register source operand, a zero-valued operand is supplied. A floating-point instruction that operates on single-precision data reads all bits of the source floating-point register. A floating-point instruction that produces a single-precision result writes all bits of the destination floating-point register.

Alpha instructions are very simple. All instructions are 32 bits long. Memory is accessed via 64-bit virtual byte addresses, using little-endian byte numbering convention. Virtual addresses, as seen by the program, are translated into physical memory addresses by the memory management mechanism. The basic addressable unit in the Alpha architecture is the 8-bit byte. Memory operations are either loads or stores. All data manipulation is done between registers.

The Alpha architecture facilitates pipelining multiple instances of the same operations because there are no special registers and no condition codes. The instructions interact with each other only by one instruction writing a register or memory and another instruction reading from the same place.

There are five basic Alpha instruction formats that contain zero, one, two or three register fields. These are: Memory, Branch, Operate, Floating-Point Operate and PALcode. The Memory format is used to transfer data between registers and memory, to load an effective address and for subroutine jumps. The Branch format is used for conditional branch instructions and for PC-relative subroutine jumps. The Operate format is used for instructions that perform integer register-to-register operations. This format allows the specification of one destination operand and two source operands. One of the sources can be a literal constant. The Floating-point Operate format is used for instructions that perform floating-point register to floating-point register operations. This format also allows the specification of one destination and two source operands. The Privileged Architecture Library (PALcode) format is used to specify extended processor functions. The PALcode is a set of subroutines that are specific to each particular Alpha operating system implementation. These subroutines provide operating-system primitives for context switching, interrupts, exceptions and memory management.

The implementations of this architecture execute instructions out of order. The processor is able to fetch, decode, rename and issue to execute up to four instructions in each cycle. There are four integer and two floating point execution units to which four instructions are issued in each cycle. The 64 KB L1 data cache accepts any combination of two loads/stores in each cycle. In total, up to 80 in-flight instructions plus 32 loads plus 32 stores can be executing at any moment.

For our tracing experiments, we have used a Compaq Alphaserver GS-160 with 16 Alpha 21264 processors running at 731 Mhz, 8Gb of main memory, and 108 GB of disk capacity. The operating system was Digital UNIX OSF1 V4.0 (Rev.1091). We used both, the C and fortran compilers, since the set of benchmarks are written in these two high level languages. We have used the C compiler from DEC, version 5.8-015, and the Compaq Fortran compiler, version 5.3-915. All programs were compiled using the -O4 -tune ev6 optimization flags which include local optimizations, recognition of common subexpressions, integer multiplication and division expansion using shifts, code motion, strength reduction and test replacement, split lifetime analysis, code scheduling, inlining of arithmetic statement functions, global optimizations that improve speed, loop unrolling, code replication to eliminate branches, inline expansion of small procedures

and specific tuning for the ev6 implementation. All these optimizations have yielded a high quality code.

2.5 BENCHMARK PROGRAMS

A very important choice in this study is the set of programs to be analyzed using simulation. We have chosen numerical and multimedia applications as the source of our benchmarks because these types of applications have both, instruction and data level parallelism that can be exploited in our ILP+DLP architecture. Numerical programs have been run traditionally on vector machines. They are characterized by being dominated by do-loop style computations that consist of several numerical expressions. Multimedia programs, as a representative of the recently appeared embedded and desktop applications, have also exposed data level parallelism, and for this reason they have also been used as benchmark programs in the recent sub-word level SIMD extensions [PW96] [NJ99] [Koh95] [MIP97].

The selected set of programs guarantee a wide range of behaviors as it includes programs from low to high vectorization degrees, as we will see in section 3.7.

2.5.1 Numerical Benchmarks

Due to the limitation on the size of the program traces, we took the Perfect Club [BCK+89] [CKPK90] and the Specfp92 [CKDK91] [SPE] programs. The Perfect Club codes were a joint effort of the CSRD at the University of Illinois, several supercomputing vendors, several universities and third party software houses, to put together a set of codes representative of the workloads of large machines, including vector supercomputers. Thus, they represent a good target for our study. Their main drawback is that, in order to make them manageable, their data sets were somewhat trimmed down. However, it is precisely this characteristic that make them a good option for us, as their trace size becomes affordable in our context. The selected programs from Perfect Club are:

- *Bdna*: The *Bdna* code performs molecular dynamics simulations of biomolecules in water. It is aimed at understanding the hydration, structure and dynamics of nucleic acids, and the role of water in the operation of biological systems.
- *Arc2d*: This program solves the Euler equations in generalized curvilinear coordinates using an implicit finite difference algorithm with approximated factorization and a diagonalization of the implicit operators.

The SPEC benchmarks [SPE] are today in widespread use for evaluating the performance of computers. They are not specifically targeted at vector machines, and they contain a fair amount of applications belonging to the scientific and engineering domains that presents data level parallelism. At the time that this study was started the specfp95 had been recently introduced. However, we decided to use the previous specfp92 because of the large size of the traces of the specfp95 benchmarks. We selected the following benchmarks from specfp92:

- *Swim256*: It is a Fortran scientific benchmark with single precision floating point arithmetic. The program solves the system of shallow water equations using finite difference approximations on a 513 x 513 grid. It is amenable to vectorization and decomposition on systems that provide such capabilities.
- *Hydro2d*: This is a vectorizable Fortran program with double precision floating point arithmetics. In this application program, from the area of astrophysics, hydrodynamical Navier Stokes equations are solved to compute galactical jets.
- *Tomcatv*: Tomcatv is a vectorizable double precision floating point Fortran benchmark. It is a mesh generation program.
- *Nasa7*: This benchmark is a group of seven kernels that carry out the following functions:
  - 2D Fast Fourier Transform,
  - Vectorized block tri-diagonal solver in the J direction for K constant planes,
  - Cholesky decomposition/substitution,
  - Inversion of 3 pentadiagonal matrices simultaneously,

- Computation of solid-related arrays,
- Gauss elimination of the matrix of wall influence coefficients,
- Emission of new vortices to satisfy boundary condition, computation of pressure, forces, etc, and 4-way unrolled matrix multiply.

2.5.2 Multimedia Benchmarks

We have selected multimedia programs from the Mediabench suite [LPMS97]. This suite represents the workload of emerging multimedia and communications systems. It is composed of complete applications coded in high-level languages. All of the applications are publicly available, making the suite available to a wider user community. Mediabench 1.0 contains 19 applications selected from available image processing, communications and DSP applications. From this set we have selected the following programs:

- *Jpeg Encode*: JPEG is a standardized compression method for full-color and gray-scale images. JPEG is lossy, meaning that the output image is not exactly identical to the input image. *Jpeg Encode* does image compression.
- *Jpeg Decode*: This application is also derived from the JPEG source code, and it does image decompression.
- *Epic*: An experimental image compression utility. The compression algorithms are based on a bi-orthogonal critically sampled dyadic wavelet decomposition and a combined run-length/Huffman entropy coder. The filters have been designed to allow extremely fast decoding without floating-point hardware.
- *Gsm Encode*: European GSM 06.10 provisional standard for full-rate speech transcoding, prI-ETS 300 036, which uses residual pulse excitation/long term prediction coding at 13 kbit/s. GSM 06.10 compresses frames of 160 13-bit samples (8 kHz sampling rate, i.e. a frame rate of 50 Hz) into 260 bits.

In summary, we selected the following benchmarks:

- **From the Perfect Club**: *Bdna* and *Arc2d*
- **From the Specfp92**: *Swim256*, *Hydro2d*, *Nasa7* and *Tomcatv*
- **From the Mediabench**: *Jpeg Encode*, *Jpeg Decode*, *Epic* and *Gsm Encode*

These ten programs will be used throughout the rest of this thesis to evaluate the performance of the proposed ILP+DLP architecture as well as the enhancements to the basic proposal.

2.6 BENCHMARK MODIFICATIONS

In order to make the study of the ILP+DLP architecture, we have had to make some changes in the original program benchmarks. These changes consist in (1) some modifications in the multimedia programs in order to get them to vectorize, (2) manual stripmining of all programs, and (3) separation of S- and D-regions.

2.6.1 Modifying Multimedia benchmarks for vectorization

Although multimedia programs have data level parallelism to be exploited by our ILP+DLP architecture, in some cases it was not exposed in such a way that the compiler could generate vector code. In those cases, some changes have been made to the multimedia benchmarks in order to get them to vectorize.

Epic has been the most heavily modified program, since we have applied from simple loop interchange techniques to a major rewrite of the *idct* algorithm following the standard specifications [GBL+98]. In *Gsm Encode*, the problem was that the compiler could not vectorize some pieces of code because some data structures involved were not local to the function code, so the compiler could not do some assumptions about them. In that case we privatizated those data structures, so the compiler could vectorize those pieces of code. *Jpeg Encode* and *Jpeg Decode* were not modified.

2.6.2 Manual Stripmining

The evaluation of the proposed ILP+DLP architecture includes a study about the size of the Vector Register File to be included. One of the factors that determines the size of the Vector Register File is the length of each vector register. Therefore, we need to generate vector code that effectively uses different vector lengths. The Convex C4 compiler only generates code for 128-element vector registers, as this is the size of the vector registers in that machine. Thus, we had to perform a manual stripmining of the vector loops for the different vector lengths that we wanted to test.

Moreover, this manual stripmining has to be done efficiently, that is, few extra operations should be added to each loop because of this code transformations. Figure 2.2 presents (a) an original loop from the *Swim256* benchmark, and (b) the stripmined version of the same loop. The variable `STRIP_LENGTH` must be set to the value of the vector length to which we want to perform the stripmine. For example, in order to obtain a version that uses 16-element vector registers we must set `STRIP_LENGTH` to 16. We can see in figure 2.2 that the `JCHECK` loop has been transformed into two nested loops (`STRIP` and `J_STRIP`) plus an additional loop (`J_STRIP`). The two nested loops carry out `NUM_STRIPS` iterations, `STRIP_LENGTH` times. The compiler will vectorize the `J_STRIP` loop, thus generating vector instructions with vector length equal to `STRIP_LENGTH`, as we needed. The final `J_STRIP` loop carries out the final iterations when `MNMIN` is not multiple from `STRIP_LENGTH`. In that case, the compiler will vectorize the final loop, using a vector length value shorter than `STRIP_LENGTH`.

This manual stripmining adds some extra instructions because of the calculations of `NUM_STRIPS` and `MODULO`. It also adds the `STRIP` and `J_STRIP` variables, and their calculations. However, it allows experimenting with different vector lengths in a machine with a fixed vector length value.

The stripmining process is carried out for all the benchmark programs, and for the different vector lengths that we want to test. After compiling the different versions we have to gather vector traces for all of them, thus obtaining different versions of the program traces, depending on the vector length used. In our case, we will use

```
DO ICHECK = 1, MNMIN
  DO JCHECK = 1, MNMIN
    PCHECK = PCHECK + ABS(PNEW(ICHECK,JCHECK))
    UCHECK = UCHECK + ABS(UNEW(ICHECK,JCHECK))
    VCHECK = VCHECK + ABS(VNEW(ICHECK,JCHECK))
  ENDDO
ENDDO
```

(a)

```
NUM_STRIPS = MNMIN/STRIP_LENGTH
MODULO = MNMIN - NUM_STRIPS * STRIP_LENGTH
DO ICHECK = 1, MNMIN
  JCHECK = 1
  DO STRIP = 1, NUM_STRIPS
    DO J_STRIP = 1, STRIP_LENGTH
      PCHECK = PCHECK + ABS(PNEW(ICHECK,JCHECK))
      UCHECK = UCHECK + ABS(UNEW(ICHECK,JCHECK))
      VCHECK = VCHECK + ABS(VNEW(ICHECK,JCHECK))
      JCHECK = JCHECK + 1
    ENDDO
  ENDDO
  DO J_STRIP = 1, MODULO
    PCHECK = PCHECK + ABS(PNEW(ICHECK,JCHECK))
    UCHECK = UCHECK + ABS(UNEW(ICHECK,JCHECK))
    VCHECK = VCHECK + ABS(VNEW(ICHECK,JCHECK))
    JCHECK = JCHECK + 1
  ENDDO
ENDDO
```

(b)

Figure 2.2 Source code of a loop (a), and the manual stripmined loop (b).

16-element, 64-element and 128-element vector registers, thus requiring three different vector traces for each program.

2.6.3 Slicing a Program into Regions

As stated in section 1.6, page 25, not all tasks can be expressed using vector instructions, and even inside a vectorizable program some parts will not be amenable to be formulated in a vector form. Therefore, from the DLP point of view, a program has two types of regions, data-parallel regions (D-regions) containing those zones that can be expressed with vector instructions, and scalar regions (S-regions), whose contents can not be expressed in vector form. A processor that exploits DLP will improve performance in those regions that are amenable to be vectorized, that is, D-regions. Therefore, in order to deeply analyze the behavior of the DLP elements of the processor, and the potential performance that can be achieved with a ILP+DLP architecture, we have separated S-regions and D-regions for each benchmark program. This separation is carried out by performing the following steps:

- First, we compile the original benchmark program on the Convex C4 machine and we obtain the compiler information about the loops that can be vectorized.
- Second, for each individual loop, we go to the source code and we put a dummy marks at the beginning and at the end of the loop. These marks are routine calls that do nothing, and do not affect the program behavior.
- Third, we compile again the modified program source, and we test that the compiler vectorizes the same loops that were vectorized in the first step.
- Fourth, we check that the assembly code generated in the first and third steps are equivalent, with the difference of two added routine calls (our marks).
- Fifth, we find where the dummy routines are situated in the program address space, in order to give this information to the simulator through an addresses file.

After this process has finished we have the same original programs, with the same vectorized regions, but before/after each D-region we have a routine call that marks

the change from a S-region to a D-regions, and from a D-region to a S-region. The simulation carries out a separate accounting of instructions, operations, etc. inside S-regions and inside D-regions. When the simulator starts a new simulation we are always on a scalar region, so all the statistics collected in those instructions are accumulated for S-regions. For each call instruction the simulator compares if the destination address corresponds to the dummy routine (our mark). If that is the case, the simulator detects a region change, so new statistic information is now accumulated for D-regions.

The statistics gathered in the frontier between two different regions are not easily assigned to an S-region or a D-region because the processor carries out out-of-order pipelined execution. In such a processor, the execution of the instructions belonging to two adjacent regions is overlapped, thus requiring a compromise on how to make the account properly. After evaluating different approaches, we found that the more accurate way is to account inside a region all those instructions that were actually executed inside that region. Therefore, in the final part of a region, when the initial instructions of the following region enter the pipeline, it may happen that some of these instructions finish their execution before those belonging to the previous region. This may happen because of the out-or-order execution. However, as instructions graduate in order, these instructions will remain ungraduated until the final instruction of the previous region graduates. At that moment, we also look at the pending instructions, and if they have finished their execution, we count those instructions as being executed inside the previous region. All in all, this aspect is not that important since the total number of overlapping operations between S-regions and D-regions, measured for every program, is always below a 4% of the total number of operations executed by the program.

2.7 THE QUALITY OF THE SCALAR CODE IN VECTOR PROGRAMS

During the last years, research efforts in compilation techniques have promoted a substantial increase in the quality of the code generated by compilers. Those new compila-

tion techniques, which have already been integrated in current compilers, have brought out the generation of a more and more efficient code.

As a consequence, we expected the quality of the scalar code generated by the vector compiler would be lower than that generated by the superscalar compiler, as was later confirmed (see sections 3.9.2 and 3.9.3), since the vector compiler was developed for a nowadays obsolete machine and can be considered as a quite rough compiler in the scalar field.

In the following chapters we will present the results about the performance that the proposed ILP+DLP architecture reaches. The performance metric is calculated by using the number of operations and the number of cycles that the processor executes. That means that the quality of the original code certainly influences the results used to evaluate the proposed architecture. Moreover, given that some of the programs have a low vectorization percentage, and according to Amdahl's Law, the larger the ratio of scalar operations executed by the program, the larger the negative influence of a low quality scalar code, and consequently, the lower the opportunity for a performance improvement. Therefore, there is no doubt that this is a drawback in the original vector programs that requires a smart solution.

The solution adopted to reach a set of programs with an acceptable scalar code consisted in generating **hybrid vector programs** starting from the original superscalar and vector programs. The hybrid version of each program is then built by merging the D-regions generated by the vector compiler and the S-regions generated by the superscalar compiler. In this way, the new version contains the best of both worlds, that is, it can be vectorized and has a good quality scalar code. Of course, these programs can still be improved as the scalar code inside D-regions was originally generated by the vector compiler, but this would involve the development of a vectorizing compiler with current scalar compilation techniques. That is, by no means, a trivial task and it is far beyond the objective of this thesis. Moreover, the performance improvement obtained in return for such an effort would be quite slight.

Therefore, our hybrid vector benchmarks are a quite good approximation to real programs compiled using an hypothetical modern vectorizing compiler. Furthermore, if our hybrid programs could be additionally improved, the results shown in this thesis would in fact be a lower bound on the real performance that would be reached by using programs compiled in a real, modern vectorizing compiler.

2.8 THE EIPC PERFORMANCE MEASURE

Comparing the execution of a scalar and a vector program in terms of instructions per cycle (IPC) poses a significant problem. As it is widely known, to execute the same code, a vector machine uses much fewer instructions than a scalar machine [QEV98b]. Therefore, using raw IPC as a performance measure would be meaningless. The solution consists in using a unique value of instructions for the SS and SSV performance measures. The meaning is that the number of instructions represents the amount of instructions needed to complete the program task. We keep this value constant for the SS and SSV, so that the final behavior comes only from the difference in the total number of cycles needed to execute the programs in both architectures. We take the total number of instructions executed in the SS architecture as the number of instructions for calculating EIPC. Therefore, the *Equivalent IPC* (EIPC) metric for each simulation of the SSV architecture is defined as:

$$SSV\ EIPC = \frac{Total\ SS\ instructions}{Total\ SSV\ Cycles} \tag{2.1}$$

The intuitive meaning of EIPC is "how well a superscalar machine should perform in order to match the performance of our proposed ILP+DLP architecture". Note that according to the previous equation the SS EIPC matches the SS IPC.

The performance study inside regions require the definition of a performance metric inside S-regions and D-regions. Formally, performance results inside D-regions and S-regions for the **hybrid programs** running in the ILP+DLP architecture are calculated as follows:

$$SSV\ EIPC\ D\text{-}reg = \frac{SS\ instr\ inside\ D\text{-}reg}{SSV\ Cycles\ inside\ D\text{-}reg} \tag{2.2}$$

$$SSV\ EIPC\ S\text{-}reg = \frac{SS\ instr\ inside\ S\text{-}reg}{SS\ Cycles\ inside\ S\text{-}reg} \tag{2.3}$$

Therefore, the overall SSV performance can also be calculated following the equation:

$$SSV\ EIPC = \frac{SS\ instr\ inside\ D\text{-}reg\ +\ SS\ instr\ inside\ S\text{-}reg}{SSV\ Cycles\ inside\ D\text{-}reg\ +\ SS\ Cycles\ inside\ S\text{-}reg} \tag{2.4}$$

The same equations are applied to the SS architecture by replacing the SSV cycles by SS cycles.

It is important to note that the EIPC metric is based in the relation between the total number of instructions executed in the SS architecture and the total number of cycles that the SSV execution lasts. Therefore, when we execute two different vector length versions of the same programs, the increase in the total number of instructions is not taken into account in the EIPC measure. Only the increase in the number of cycles is considered, meaning that a lower EIPC will be produced. Therefore, the performance behavior is made obvious.

3

SCALAR AND VECTOR ISAS COMPARISON

Summary

This chapter studies the scalar and vector architectures from the instruction set point of view. It presents data on the instruction level characteristics of the selected benchmarks when compiled in both a superscalar and a vector platform. We also make an analysis of the scalar and data-parallel regions (S- and D-regions) and introduce the ISA characteristics of the hybrid versions of the benchmarks that will be used in the rest of this thesis.

3.1 INTRODUCTION

Architectures have traditionally been evaluated by using performance metrics, such as elapsed time or number of instructions executed per cycle, on complete benchmark programs. This type of metrics allows extracting interesting information about how well the architecture is able to execute the program from the user's point of view. Although we will also analyze performance metrics in the following chapters, we will first introduce a detailed analysis about the low level characteristics of programs compiled on a superscalar processor as well as on a vector processor. There is a double purpose in this study. On one side, we want to analyze the different nature of scalar and vector instruction sets. Vector architectures have some characteristics that result in certain advantages when instructions are executed. This will lead us to conclude that it is worth exploring the possibility of including vector instructions (plus a vector functional unit) in a current superscalar architecture. On the other side, this analysis allows a deep knowledge about low level details of the benchmarks, and the performance deficiencies detected using performance metrics can be better explained.

For example, given a vectorizable piece of code, a vector architecture generates a lower number of instructions than a superscalar architecture. The reason is the higher semantic level of the vector instructions, which implies that more than one operation is carried out in each instruction. When this piece of code is executed in a superscalar processor, more instructions must be fetched and decoded in each cycle, and a larger instruction window is needed. Only by knowing how many instructions the superscalar and vector processors execute, can we analyze if these values are different enough to conclude that the fetch and decode width or the instruction window depth, are performance bottlenecks for a certain superscalar configuration.

This chapter will look at the low level characteristics of our set of benchmarks. In particular, we are interested in the following topics:

- **Benefits of a Vector ISA:** It is a study about the characteristics of vector instruction sets that allow better instruction fetch bandwidth, memory system per-

formance (in terms of both memory latency and memory bandwidth), and datapath control.

- **Basic Block, Instruction and Operation Counts and Data Types:** These are measures that support the previous topic and allow analyzing the differences in terms of instructions and operations between both ISAs. It also looks at the data types used by the programs.
- **Vector Characterization:** It analyzes the vectorization percentage of the different programs as well as the distribution of vector lengths and vector strides, and the vector mask and vector first executions.
- **Influence of the Vector Length:** This is an important feature because it allows determining the length of the vector registers in the architecture we propose, as it shows the evolution in total number of instructions, operations and processor-memory traffic as the vector length decreases.
- **Analysis by Regions:** As stated in the previous chapter, we have separated scalar and vector regions in our analysis in order to be able to identify the performance behaviors in both zones. Scalar regions (S-regions) will perform the same in both architectures, while data-parallel regions (D-regions) will be improved in the vector programs. In this section, we measure the instruction level behaviors inside S- and D-regions.
- **Hybrid Benchmarks:** The section introduces the hybrid versions of the programs that will be used during the rest of this thesis. Each hybrid benchmark is built as a mixture of the original scalar and vector versions of each program. We will use the best of both worlds in order to predict the behavior of future ILP+DLP architectures running vectorizable numerical and multimedia codes.

3.2 BENEFITS OF VECTOR ISA

Exploiting data level parallelism has many advantages that can be classified in three areas: instruction fetch bandwidth, memory system performance (latency and band-

width), and datapath control. This section will outline the benefits of using a vector instruction set in each of these areas.

Instruction fetch bandwidth. The main difference between a vector and a scalar instruction is that the vector instruction has a higher semantic content in terms of the operations specified. This semantic content also carries the notion of parallelism, since each individual operation of a vector instruction is known to be independent of all other operations of the same instruction. This difference results in a myriad of related advantages [QEV98a]. First, to perform a given task, a vector program executes many fewer instructions than a scalar program, since the scalar program has to specify many more address computations, loop counter increments and branch computations, which are typically implicit in vector instructions (section 3.4 provides quantitative support for this claim). As a direct consequence, the instruction fetch bandwidth required, the pressure on the fetch engine and the negative impact of branches are all three reduced in comparison to a scalar processor. Besides, a relatively simple control unit is enough to dispatch a large number of operations in a single cycle, whereas a superscalar processor devotes an always increasing part of its area to manage out-of-order execution and multiple issue. This simple control, in turn, can potentially yield a faster clocking of the whole datapath.

Memory system performance. Due to the ever increasing gap between memory and cpu speed, current superscalar micros need increasingly large caches to keep up performance. Nonetheless, despite out-of-order execution, non-blocking caches and prefetching, superscalar micros do not make an efficient use of their memory hierarchies. The main reason for this inefficient use comes from the inherently predictive model embedded in cache designs. Whenever a line is brought from the next level in the memory hierarchy, we do not know whether all data will be needed or not. Moreover, it is very uncommon for superscalar machines to sustain the full bandwidth that their first level caches can potentially deliver. Since load/store instructions are mixed with computation and setup code, dependencies and resource constraints prevent a memory operation from being launched every cycle.

In contrast, in the vector style of accessing memory every single data item requested by the processor is actually needed. There is no implicit prefetching due to lines. Moreover, the information on the pattern used to access memory is conveyed to the hardware through the stride information and it can be used to improve memory system performance [VLL+92] [VLPA95] [PVAL95].

- **Memory Latency**: When it comes to memory latency, a vector memory instruction can amortize long memory latencies over many different elements. By using some ILP techniques coupled with a vector engine, up to 100 cycles of main memory latency can be tolerated with a very small performance degradation [Esp97].
- **Memory Bandwidth**: Regarding memory bandwidth, a vector machine can make a much more effective usage of whatever amount of bandwidth it is provided with. While a superscalar processor requires extra issue slots and decode hardware to exploit more ports to the first level cache, a vector machine can request several data items with a single memory address. For example, when doing a stride-1 vector memory access, a vector machine need not send every single address to the memory system. Simply sending every N^{th} address, a bandwidth of N words per cycle can be achieved [CEV98].

Datapath Control. In order to scale current superscalar performance up to, say, twenty instructions per cycle, an inordinate amount of effort is needed. The dispatch window and reorder buffers required for such machine are very complex. The wakeup and select logic grows quadratically with the number of entries, so the larger the window the more difficult it is to build such an engine [PJS97]. If superscalars with 4-wide dispatch logic barely sustain 1 instruction per cycle, a superscalar machine that sustained say 20 operations per cycle seems not feasible.

On the other hand, the fact that the individual operations in a single vector instruction are independent allows a more efficient execution: once a vector instruction is issued to a functional unit, it will use it with effective work for many cycles. For this reason, a vector engine can be easily scaled to higher levels of parallelism by simply replicating the functional units and adding wider paths from the vector registers to the functional

Program	Vector ISA	Scalar ISA
Swim256	2.59	6.75
Hydro2d	11.18	263.25
Nasa7	16.85	24.18
Tomcatv	19.95	19.91
Bdna	41.81	33.71
Arc2d	4.56	25.83
Jpeg Decode	6.25	8.51
Epic	2.92	2.00
Jpeg Encode	11.67	29.69
Gsm Encode	15.21	28.13

Table 3.1 Total number of dynamic basic blocks executed in the vector and scalar machines. Both columns are in millions.

units. All this without increasing at all the complexity or the pressure on the decode unit. The semantic contents of the vector instructions already include the notion of parallel operations.

It is important to note that to be able to make use of a vector instruction, compilers must include vector compilation techniques that can expose data level parallelism, so that programs can be vectorized. However, due to the great efforts that have been made for vector supercomputers and their compilers [Wol90] [ZC91], the field of vector compilation has reached maturity and superscalar compilers can take advantage of this.

3.3 BASIC BLOCK DISTRIBUTION

Table 3.1 presents the total number of basic blocks executed in the superscalar and vector machines for the set of benchmarks. The interesting aspect of this table is the total number of basic blocks executed in the superscalar machine when compared to the same value in the vector machine. Scalar programs execute a larger number of basic blocks for almost all programs. One of the reasons is the predicated execution that the vector processor carries out through the *vector mask* execution, as we will discuss in a later section. The other reason, which is more important, is that the use of vector instructions allows specifying multiple operations on independent data, thus reducing the number of loop iterations. For example, given the loop in figure 3.1 (a),

which makes an operation over the 1024 elements of two vector data structures, let us compile it on a superscalar and on a vector platform. In the superscalar platform, see 3.1 (b), it is necessary to initialize a loop counter to 1024 that will then be decremented by 1 in each loop iteration, until it reaches 0. The superscalar machine would need 1024×3 index increments, 1024 loop counter decrements and 1024 conditional branches. Executing the same loop in a vector platform with vector length 128 would require eight loop iterations, see figure 3.1 (c). Each iteration would execute four vector instructions (load, load, add and store) using vector length 128, three index increment instructions, one loop counter decrement instruction and one conditional branch instruction. The total number of index increments would be $8 \times 3 = 24$, as well as 8 loop counter decrements and 8 conditional branches. In short, the superscalar platform executes the basic block of this loop 1024 times while the vector platform executes it only 8 times.

The example shown in figure 3.1 is rather trivial. In a real case, the number of iterations is not likely to be an exact multiple of the vector length. In that case, the compiler must introduce certain additional basic blocks that are used to calculate how many iterations must be performed, and they increase the total number of basic blocks that are executed for this loop. In the example above, if the number of loop iterations were 1028, the vector processor would execute eight iterations using vector length 128 and one iteration using vector length 4.

Although the number of basic blocks executed is an interesting measure, it does not provide much information on its own. In order to acquire a more accurate knowledge of the computational load of the benchmark programs, it is necessary to measure how many instructions are included in each basic block. The following sections present the number of instructions and operations executed by each program, which allows a more detailed analysis of the scalar and vector instruction sets.

3.4 INSTRUCTION BREAKDOWN

As already mentioned, vector instructions have a higher semantic content in terms of the operations specified. The result is that, to perform a given task, a vector program

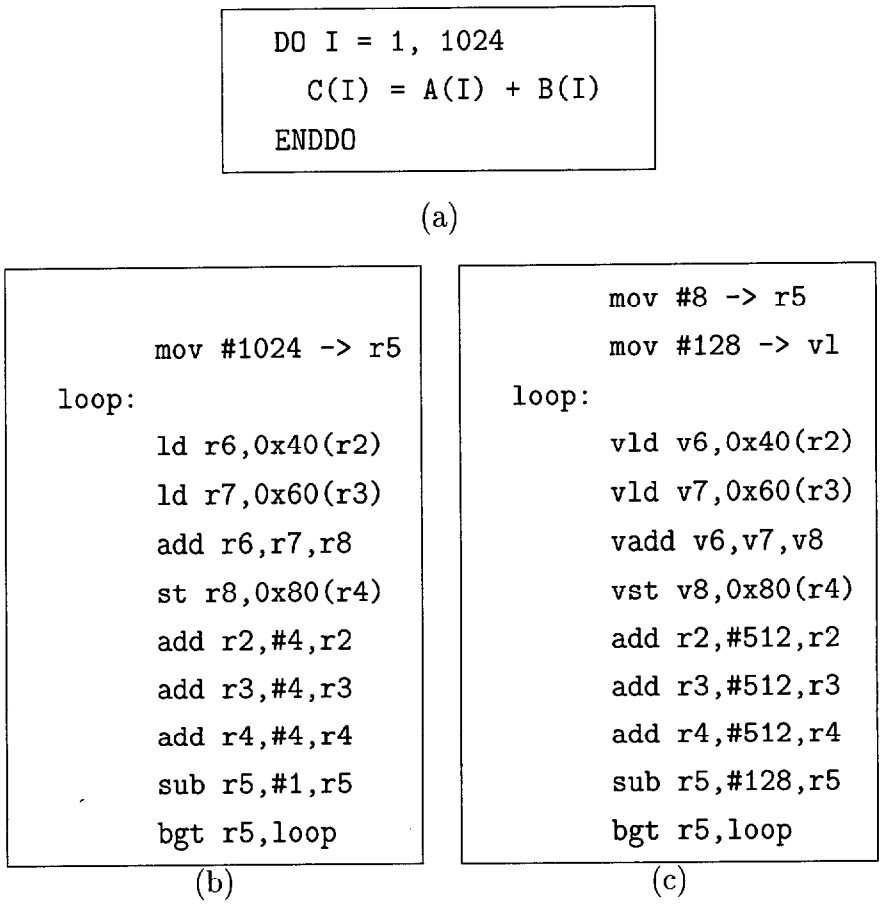


Figure 3.1 Source code of a basic loop (a), and assembler pseudo-code of loop compiled in a superscalar platform (b) and in a vector platform (c).

executes many fewer instructions than a scalar program, since the scalar program has to specify more address calculations, loop counter increments and branch computations, which are typically implicit in vector instructions. The net effect of vector instructions is that, in order to specify all the computations required for a certain program, much fewer instructions are needed.

Table 3.2 presents the total number of instructions executed in the superscalar ISA and in the vector ISA for the ten benchmark programs, in millions. Columns 2 and 3 contain the total number of instructions executed in the superscalar and vector platforms, respectively. Columns 4 and 5 are the breakdown of column 3 into scalar instructions

Program	Scalar ISA	Vector ISA		
	Total	Total	Scalar	Vector
Swim256	9438.59	176.15	101.31	74.84
Hydro2d	4671.75	215.70	178.16	37.54
Nasa7	5542.03	1310.92	1253.21	57.71
Tomcatv	897.66	152.28	145.45	6.83
Bdna	1628.97	504.60	497.41	7.19
Arc2d	4905.61	140.70	101.13	39.57
Jpeg Decode	154.00	52.75	52.19	0.55
Epic	35.58	16.89	15.89	0.99
Jpeg Encode	325.02	90.96	80.48	10.47
Gsm Encode	669.24	102.61	101.77	0.83

Table 3.2 Total number of instructions executed by each program in the scalar ISA and in the vector ISA. For the vector ISA the instructions are also classified into scalar and vector. All columns are in millions.

and vector instructions. As observed, the differences between columns 2 and 3 are huge. The scalar versions of the programs execute from 2 up to 53 times more instructions than the vector versions.

Regarding the breakdown into scalar and vector instructions in the vector architecture, although the vector instructions of the programs carry out most of the computations (as we will see in the following sections), they execute very few instructions. Obviously, the results depend on the vectorization percentage of each program. Although several compiler optimizations (loop unrolling, for example) can be used to lower the overhead of typical loop control instructions in superscalar code, vector instructions are inherently more expressive.

Having vector instructions allows a loop to do a task in fewer iterations. This implies fewer computations for address calculations and loop control, as well as less instructions dispatched to execute the loop body itself. As already discussed in section 3.2, the direct consequence of executing less instructions is the reduction of the instruction fetch bandwidth required, the pressure on the fetch engine and the negative impact of branches, as well as a simpler control unit.

In tables 3.3, 3.4 and 3.5, we have included the basic instruction distribution for the ten benchmark programs. For the vector programs, we have classified all the instructions

Program	Scalar Instructions					
	Add	Mul	Ld	St	Control	Total
Swim256	84.76	0.03	3.60	5.08	7.82	101.31
Hydro2d	105.09	0.92	32.27	19.38	20.47	178.16
Nasa7	634.43	92.07	311.67	110.16	104.84	1253.21
Tomcatv	110.44	0.20	13.99	0.57	20.23	145.45
Bdna	264.55	48.49	101.65	34.68	40.81	497.41
Arc2d	66.18	0.55	22.55	4.59	7.24	101.13
Jpeg Decode	30.51	2.60	9.61	4.46	4.99	52.19
Epic	7.91	0.36	3.24	1.94	2.43	15.89
Jpeg Encode	45.10	0.01	9.61	9.53	10.82	80.48
Gsm Encode	63.44	5.84	14.55	6.89	11.04	101.77

Table 3.3 Breakdown of scalar instructions for the whole vector programs. All columns are in millions.

Program	Vector Instructions					
	Add	Mul	Dyadic	Ld	St	Total
Swim256	21.23	10.86	11.05	21.26	10.20	74.84
Hydro2d	12.73	8.14	0.82	11.68	4.14	37.54
Nasa7	10.35	12.51	9.56	17.90	7.37	57.71
Tomcatv	2.34	1.83	0.15	1.75	0.74	6.83
Bdna	1.42	1.93	1.04	1.70	1.07	7.19
Arc2d	6.70	11.36	2.52	14.04	4.93	39.57
Jpeg Decode	0.29	0.08	0.08	0.05	0.04	0.55
Epic	0.49	0.20	0.00	0.28	0.01	0.99
Jpeg Encode	2.33	0.41	2.40	4.97	0.34	10.47
Gsm Encode	0.37	0.03	0.14	0.21	0.07	0.83

Table 3.4 Breakdown of vector instructions for the whole vector programs. All columns are in millions.

in two major groups: scalar (table 3.3) and vector (table 3.4). This classification is not needed in superscalar programs as all instructions executed in these programs are scalar. Each group has been further broken down in a total of six categories. These categories are: **Add**, **Mul**, **Dyadic**, **Ld**, **St** and **Control** instructions. In the vector group there is no **Control** category, as control instructions are scalar. Similarly, in the scalar group there is no **Dyadic** category, as these are vector only instructions. **Ld** and **St** categories in the vector group include, respectively, the **gather** and **scatter** instructions. The **Mul** category includes all types of instructions that can only be executed in a general purpose functional unit, that is, it includes all multiplications, divisions and square roots. The **Add** category includes all other arithmetic operations

Program	Instructions					
	Add	Mul	Ld	St	Control	Total
Swim256	3890.28	1750.01	2565.41	1168.04	64.83	9438.59
Hydro2d	1882.06	631.73	1469.83	398.29	289.83	4671.75
Nasa7	1699.81	982.82	2093.57	646.09	119.72	5542.03
Tomcatv	347.07	162.67	274.47	88.58	24.85	897.66
Bdna	609.37	362.06	450.59	173.27	33.66	1628.97
Arc2d	1684.41	1072.53	1524.59	543.87	80.19	4905.61
Jpeg Decode	123.50	0.49	16.29	7.75	5.96	154.00
Epic	18.80	3.08	9.59	0.75	3.35	35.58
Jpeg Encode	179.05	18.36	88.43	14.98	24.18	325.02
Gsm Encode	507.83	33.58	83.03	21.73	23.05	669.24

Table 3.5 Breakdown of instructions for the whole superscalar programs. All columns are in millions.

not included in the **Mul** category, that is, additions, subtractions, shifts, logicals, etc. In all these tables, the last column contains the total number of instructions executed in that group, recalled from table 3.2 in this section.

These tables allow us to study the instruction distribution in scalar and vector programs. In table 3.3, most of the scalar instructions in vector programs belong to the **Add** category. This is not surprising, since scalar code found in vectorizable programs mostly consists of address computations (additions and shifts) and loop control instructions (decrements and comparisons). Vector instructions do not follow a clear pattern (see table 3.4); in some programs, the majority of the vector instructions are in the **Add** category while in others are found in the **Ld** or **Mul** categories. Once more, in the scalar programs, the majority of the instructions are found in the **Add** category, followed by the **Ld** category, as observed in table 3.5.

3.5 OPERATION DISTRIBUTION

A more accurate comparison between the superscalar and vector ISAs can be obtained by considering the total number of operations performed. As already mentioned, the overhead reduction due to the semantic content of vector instructions should result in a lower number of operations executed in the vector model. Table 3.6 shows the total number of operations executed on each platform, for each program. Columns 2 and

3 in this table present the total number of instructions executed in the superscalar and vector ISAs, respectively. As expected, the total number of operations is greater in the superscalar platform than in the vector machine, for all programs. The ratio of superscalar operations to vector operations, shown in column 6 in this table, is favorable to the vector model by factors that normally go from 1.05 up to 1.25, being as large as 4.14 for program *Gsm Encode.*

The reason of the reduction in the number of operations in the vector programs comes from the higher semantic content of vector instructions. As discussed in section 3.2, 55, one vector instruction involves several operations (as many as the vector length indicates) over a set of independent data. Therefore, it is not necessary to include specific instructions to decrement loop counters, evaluate conditions and branch conditionally.

For example, let us consider a loop moving 256 words of data from array A to array B. In a superscalar ISA, a typical loop would consist of about 5 instructions: a load, a store, an addition to increment the address pointer, a subtraction to decrement the loop counter and a compare-and-branch instruction. To move 256 words, the loop would execute $256 \times 5 = 1280$ instructions (or operations). On the other hand, a vector machine, would also have the same 5 instructions in the loop. But, the load and store would be vector instructions, each one responsible of moving 128 elements. Thus, the vector version of the loop would require just two iterations and, as a whole, it would have executed about 10 instructions, or 518 operations, to perform the same task.

To get an idea about the number of reduced operations, let us consider, for example, *Swim256.* In vector mode, it requires 8230.99 million operations, while in superscalar mode it requires 9438.59 million instructions. The difference between these two amounts, that is 1207.60 million scalar instructions, is the extra overhead that the superscalar machine has to pay due to the larger number of loop iterations it performs.

Columns 4 and 5 from table 3.6 present the number of scalar and vector operations executed by the vector programs. These values show that the majority of the programs execute much more vector than scalar operations. These values will be used in the following section in order to calculate how vectorizable the programs are.

Program	Scalar ISA	Vector ISA			Ratio
	Total	Total	Scalar	Vector	
Swim256	9438.59	8230.99	101.31	8129.68	1.14
Hydro2d	4671.75	4085.72	178.16	3907.56	1.14
Nasa7	5542.03	5139.91	1253.21	3886.70	1.07
Tomcatv	897.66	848.31	145.45	702.85	1.05
Bdna	1628.97	1497.31	497.41	999.90	1.08
Arc2d	4905.61	4089.32	101.13	3988.18	1.20
Jpeg Decode	154.00	133.85	52.19	81.65	1.15
Epic	35.58	28.39	15.89	12.49	1.25
Jpeg Encode	325.02	268.78	80.48	188.29	1.21
Gsm Encode	669.24	161.42	101.77	59.64	4.14

Table 3.6 Total number of operations executed by the programs. Columns 4 and 5 are the number of scalar and vector operations executed by the vector programs. All columns are in millions.

Program	Vector Operations					
	Add	Mul	Dyadic	Ld	St	Total
Swim256	1891.00	946.09	2200.43	2061.73	1030.41	8129.68
Hydro2d	1294.86	826.84	168.21	1199.57	418.06	3907.56
Nasa7	716.27	683.89	659.53	1273.53	553.46	3886.70
Tomcatv	214.52	175.47	26.01	205.44	81.40	702.85
Bdna	169.75	233.50	258.31	207.91	130.42	999.90
Arc2d	631.41	1075.83	483.62	1329.15	468.16	3988.18
Jpeg Decode	37.82	10.59	21.18	6.63	5.41	81.65
Epic	5.45	2.65	0.00	4.29	0.09	12.49
Jpeg Encode	51.69	15.57	63.94	49.53	7.54	188.29
Gsm Encode	16.08	0.84	23.59	15.36	3.75	59.64

Table 3.7 Breakdown of vector operations for the whole vector programs. All columns are in millions.

We will conclude this section by analyzing the operation distribution across the different categories defined in the previous section. Although we should show measurements regarding scalar and vector operations for the vector programs, and scalar operations for the scalar programs, in fact, one scalar instruction carries out just one operation, so that measurements about scalar operations and instructions match and will be recalled from the previous section. Therefore, the only new data in this section, shown in table 3.7, present the number of vector operations executed in vector programs.

For most of the programs, the number of scalar operations in vector programs, shown in table 3.3, hardly influences the total number of operations executed in each category,

Operation type	No Size	Integer				Floating Point	
		8 bits	16 bits	32 bits	64 bits	32 bits	64 bits
Memory	0	10.67	0.22	130.32	9437.03	-	-
Arithmetic	54.99	0.86	0.53	892.34	1082.41	0.004	12032.24
Other	210.83	0	0	0	0	0	0

(a) Numerical Programs

Operation type	No Size	Integer				Floating Point	
		8 bits	16 bits	32 bits	64 bits	32 bits	64 bits
Memory	0	22.65	29.53	104.76	0.93	-	-
Arithmetic	8.02	1.42	3.17	336.79	47.19	2.92	5.71
Other	28.61	0.70	0	0	0	0	0

(b) Multimedia Programs

Table 3.8 Basic operations counts for the different data types for the numerical and multimedia programs. (All columns are in millions).

so we can make a raw comparison between the data shown in tables 3.5 and 3.7. Note that, one **Dyadic** instruction executes two operations (one add operation plus one mul operation). Therefore, in the **Dyadic** category, a half of the operations are add-type and the other half are mul-type. Keeping this in mind, we realize that, in general, vector programs execute less **Add**, **Ld**, **St** and **Control** operations than scalar programs. However, more **Mul** operations are executed in vector programs.

3.6 DISTRIBUTION OF DATA TYPES

Table 3.8 presents the distribution of data types that are used by (a) the numerical programs, and (b) the multimedia programs for the execution of the memory instructions, arithmetic instructions, and the rest. Column 2 has a special meaning, as it presents those instructions that are considered to be carried out with "No Size", for example branch instructions, subroutine calls and some logicals. We can also see that memory instructions can not be separated into integer and floating point. The reason is that the Convex C4 architecture does not have integer and floating point registers that we could use to make that separation. Rather, scalar registers (S registers) and vector registers (V registers) are used, and they can contain either an integer or a floating point data.

We can see in table 3.8 that the two sets of benchmarks have different characteristics regarding the data size. For example, numerical programs mainly move 64-bit elements with the memory system, while in multimedia benchmarks the majority of the memory operations are carried out with 32-bit elements. Accordingly, numerical programs execute the majority of their arithmetic operations with 64-bit elements, and multimedia programs executes its larger amount of arithmetic operations with 32-bit data.

Finally, multimedia benchmarks execute many operations with 8-bit and 16-bit data, when compared to the numerical programs. Therefore, these multimedia benchmarks have an intrinsic source of parallelism for short data sizes that could be exploited by using some kind of sub-word level parallelism [CEV99] [JTVW01].

3.7 VECTOR CHARACTERIZATION

Vector ISAs have certain inherent characteristics that do not exist in superscalar ISAs. These characteristics include, for example, the vectorization percentage, that measures how vectorizable the programs are, the effective usage of the vector length, the use of the vector stride, the execution under vector mask, or the execution under vector first. In this section we will deeply analyze these characteristics as they will be of help in order to understand the performance behavior of these programs in the following sections.

3.7.1 Vectorization Percentage and Average Vector Length

The first interesting question when studying the vector characterization of a set of programs is how vectorizable programs are, that is, the percentage of the task that is amenable to be expressed using vector instructions. We have called this measure the **vectorization percentage**, and it is defined as the ratio between the number of vector operations and the total number of operations performed by the program. Table 3.9 presents the basic statistics that allow us to compute the vectorization percentage. Columns 2 and 3 contain the total number of instructions issued by the decode unit, broken down into scalar and vector instructions. Column 4 contains the number of

Program	# instructions		# vector operations	Vect %	Avg. VL
	Scalar	Vector			
Swm256	101.31	74.84	8129.68	98.77	109
Hydro2d	178.16	37.54	3907.56	95.64	104
Nasa7	1253.21	57.71	3886.70	75.62	67
Tomcatv	145.45	6.83	702.85	82.85	103
Bdna	497.41	7.19	999.90	66.78	139
Arc2d	101.13	39.57	3988.18	97.53	101
Jpeg Decode	52.19	0.55	81.65	61.00	147
Epic	15.89	0.99	12.49	44.01	13
Jpeg Encode	80.48	10.47	188.29	70.05	18
Gsm Encode	101.77	0.83	59.64	36.95	71

Table 3.9 Basic operations counts for the set of benchmarks on the vector machine (Columns 2-4 are in millions).

operations performed by vector instructions. Each vector instruction can perform many operations (up to 128), hence the distinction between vector instructions and vector operations. The fifth column is the percentage of vectorization calculated as column 4 divided by the addition of columns 2 and 4. Finally column 6 presents the average vector length used by vector instructions, that is, the ratio between vector operations and vector instructions (columns 4 and 3, respectively).

The analysis of the vectorization percentage shows that our benchmark programs are either highly or moderately vectorizable, considering that a highly vectorizable program has a vectorization percentage over 70%. Among highly vectorizable programs we have *Swim256*, *Hydro2d*, *Nasa7*, *Arc2d* and *Jpeg Encode* programs, while *Bdna*, *Jpeg Decode*, *Epic* and *Gsm Encode* are considered to be moderately vectorizable.

Another interesting point extracted from this table is the average vector length observed in the programs. It is not strongly related to the percentage of vectorization and, in some cases, it is greater than 128, the theoretical maximum vector length. The reason is that dyadic vector instructions execute twice as many operations as the vector length, which in fact makes the ratio between vector operations and vector instructions greater than 128 if the program executes an important amount of dyadic instructions and its vector length is close or equal to the maximum. Of course, every program that executes dyadic instructions is affected by this increase in the average vector length even though

it does not exceed the maximum vector length. The higher the proportion of dyadic instructions executed, the larger the increase in the average vector length.

3.7.2 Vector Length Distribution

Vector execution is based on performing a certain operation specified in one instruction over a large amount of independent data. The amount of data of each instruction is dynamically specified with the value of the Vector Length register (VL). The latency of the operation being carried out is then amortized across all VL elements. Therefore, the larger the vector length, the better the performance. Fig. 3.2 presents the vector length distribution for the set of programs when varying the maximum vector length that the vector processor can use, i.e. the length of each vector register in the vector register file. We have used vector lengths of 128, 64, 32 and 16 elements.

As we can see, the vector length distributions follow several patterns. *Swim256*, *Tomcatv*, *Bdna*, *Arc2d* and *Jpeg Decode* have most of their vector lengths concentrated around 128. As the vector length decreases, each access must be divided into two smaller accesses. This causes an increase in the number of accesses with the maximum vector length, as the vector length decreases. Furthermore, the percentage contribution of small vector lengths decreases.

Hydro2d has a single dominant vector length, which is the number of grid points used in the z-direction of the problem, i.e. 102. When the maximum vector length is 128, a single vector instruction can carry out 102 operations in a single go. When the maximum vector length is set to 64, an instruction with 102 operations must be carried out by using two vector instructions; one of them with vector length 64 and the other with vector length 38. That is the reason of the step in figure 3.2 for the VL 64 plot.

When the vector length is again divided by two, each instruction with vector length 64 is carried out with two instructions with vector length 32, and the instruction with vector length 38 is carried by an instruction with vector length 32 plus another with vector length 6. This is the explanation of the step in the VL 32 plot. For the VL

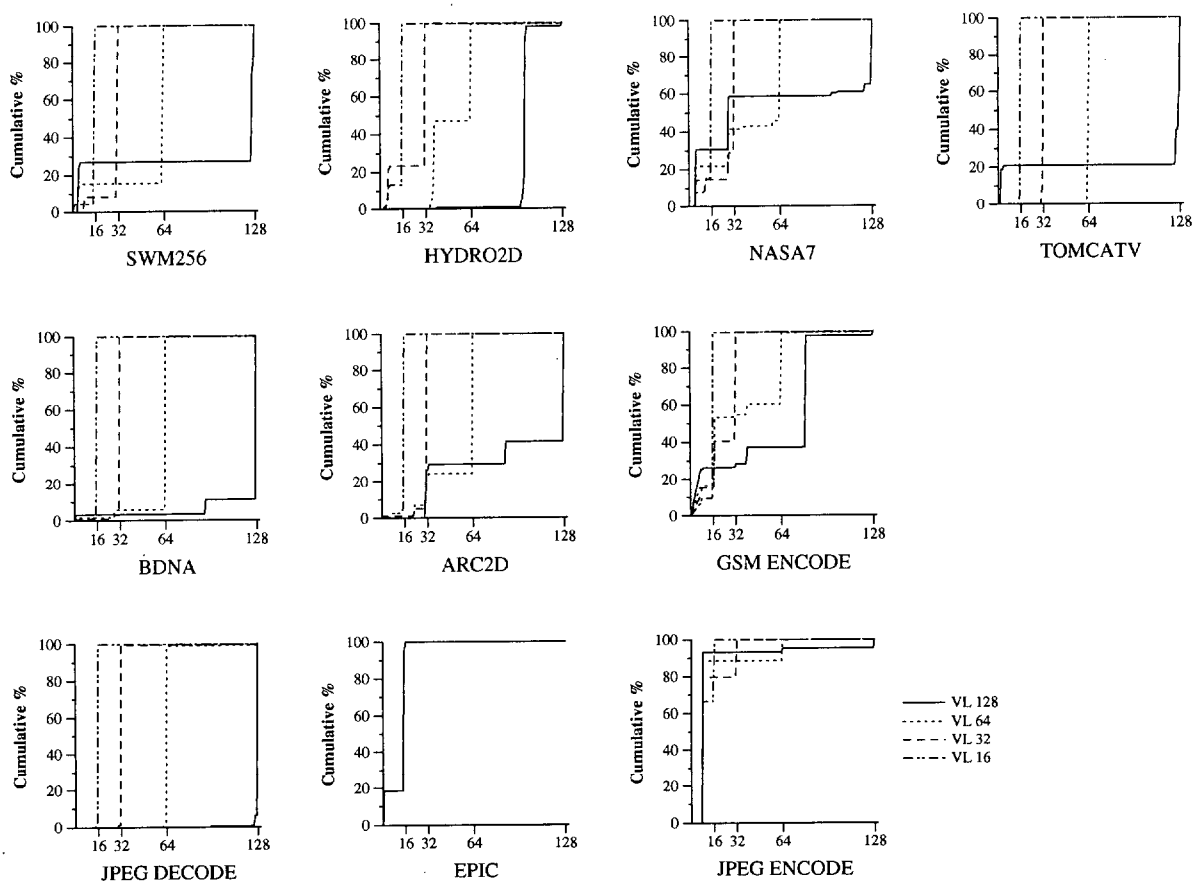


Figure 3.2 VL Distribution.

16 plot, the same phenomenon happens and, in this case, the vector instructions with vector length 6 remain the same.

Nasa7 and *Gsm Encode* have a distribution that follows a staircase, having several dominant vector lengths. In the same way, as the vector length decreases, the number of instructions that use the maximum vector length increases, and the relative importance of small vector lengths decreases. In *Jpeg Encode*, the dominant vector length is 8, although there is a minor use of other larger values. In this case, the effect is the same but it is concentrated in top left corner of the graph, as the percentage of instructions with the dominant vector length is very large and only those instructions with large VL values are affected by the decrease in the maximum vector length. *Epic* presents as unique vector length 16, so that there is no difference between the four plots. In this case, decreasing the maximum vector length does not introduce any change. All

Program	Stride Value							Gather/ Scatter
	1	2	3	4	5	8	≥ 16	
Swm256	99.74	-	-	-	-	-	0.26	-
Hydro2d	99.64	-	-	-	-	-	0.26	0.10
Nasa7	35.60	15.41	0.0005	11.96	1.24	-	35.79	0.0008
Tomcatv	100	-	-	-	-	-	-	-
Bdna	86.84	-	0.001	13.16	-	-	-	0.002
Arc2d	78.80	-	-	-	-	-	12.15	9.05
Jpeg Decode	68.83	12.52	18.84	-	-	-	-	0.01
Epic	70.20	-	-	-	-	-	29.80	-
Jpeg Encode	55.92	-	11.93	-	-	32.15	-	-
Gsm Encode	100	-	-	-	-	-	-	-

Table 3.10 Percentage of vector memory operations carried out with each stride value, including gather and scatter number of operations.

these data suggest that, even among vectorizable programs, the utilization of the vector registers varies a lot.

3.7.3 Vector Stride Distribution

Another important metric in the vector programs is the vector stride that is used in the vector memory accesses. The vector stride is the amount of bytes that separate two consecutive elements of a vector memory access. When the memory hierarchy consist of a cache hierarchy, as it is in the memory hierarchy that we propose, the best performance results are obtained for stride-1 vector memory accesses and it results in a better utilization of the memory bandwidth. In a stride-1 vector memory access the full cache line is delivered to the processor. A stride-2 vector memory access only uses a half of the elements in the cache line, and so on. When the stride is larger than the number of elements in a cache line, each cache line accessed only provides one element, thus minimizing the benefits of exploiting the bandwidth that the memory can deliver. Gather and scatter memory instructions access to memory elements that are not separated a fixed stride. Therefore, they behave the same as large strided memory accesses.

Table 3.10 presents the vector stride distribution for the set of benchmarks under study. Some of the benchmarks, like *Swim256*, *Hydro2d*, *Tomcatv* and *Gsm Encode*, execute

the majority of their memory accesses with stride equal 1. In this case, programs directly benefit from the memory bandwidth, and an increase in this characteristic will probably translate in an improved performance. Other programs, however, execute memory accesses with stride-1 and some other relatively short strides. This is the case of *Jpeg Decode* and *Bdna*, which execute their memory accesses with stride equal 1, 2, 3 and 4. In this case there will be a performance loss that depends on the vectorization percentage of the program. Finally, the worst behavior will appear in the rest of the programs, as they execute a large fraction of their memory operations with a large stride. This is the case of *Jpeg Encode*, *Nasa7*, *Arc2d* and *Epic*, and especially in *Nasa7* and *Arc2d* because of their high vectorization percentage. These programs do not make a good use of the available memory bandwidth because of their large strides. In a large fraction of their vector memory accesses the memory only provides one data in each cycle. As we will see in later chapters, these programs will be limited by the memory hierarchy.

3.7.4 Vector First Capability

A capability in the Convex C4 processor that we would like to mention is the Vector First **VF** facility, which allows specifying the first element in the vector register on which the instruction will be executed. That is, an instruction executes VL operations starting at element VF. This facility avoids having to reload data in the cases of data reuses as those presented in figure 3.3(a). In these cases, instead of executing two load instructions for matrix B (for positions I and I+1, as presented in Fig. 3.3(b)), only one load instruction is needed. Fig. 3.3(b) shows the assembly code without vector first. Every add instruction involves two vector load instructions, which is redundant. In Fig. 3.3(c), using vector first, the same data can be reused in the loop body just using the appropriate vector first value, so just one vector load is needed for each add instruction. [Note that the notation "^v0" means execution under vector first].

Table 3.11 presents the distribution of the vector first values for our set of benchmarks. This table shows the total number of instructions and operations carried out under vector first, the percentage of total operations executed under vector first, and the respective percentages of operations that have been executed with vector first equal

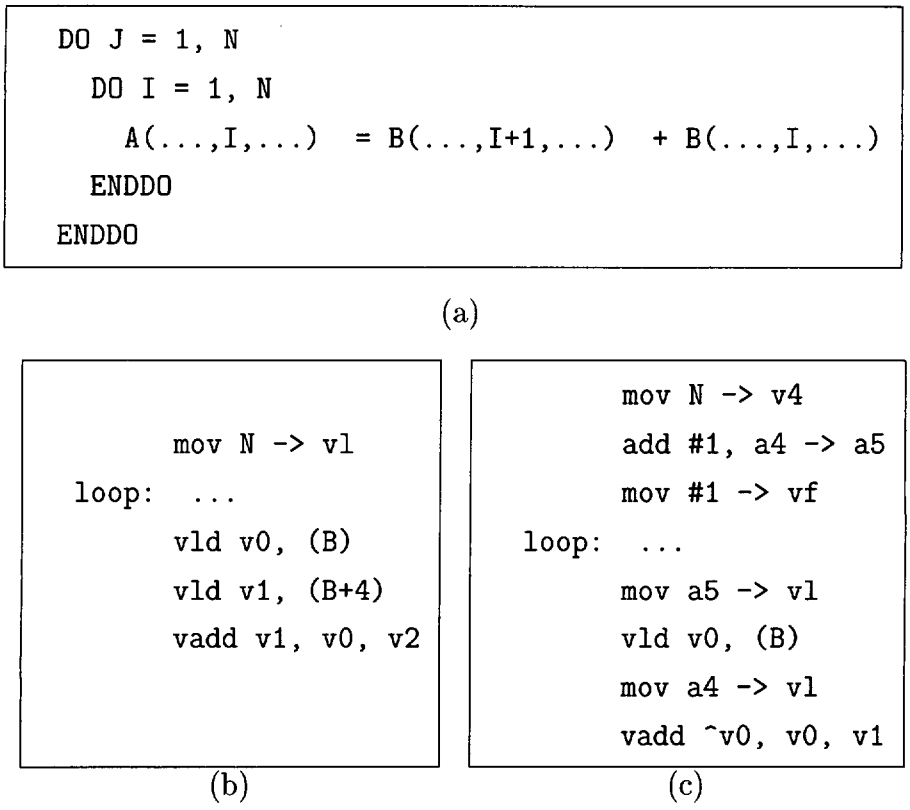


Figure 3.3 Typical vector loop at hydro2d benchmark. (a) Source code for a vector loop with data reuse of distance 1. (b) Assembly code without using vector first facility, with add involving two load instructions. (c) Assembly code using vector first so that every data must be loaded just once.

to 1, 2 or other values. The compiler is able to use the vector first in benchmarks *Swim256*, *Hydro2d*, *Tomcatv* and *Epic*. In some of these programs, the percentage of operations executed under vector first is rather important, and this fact must be considered as an additional reason for the decrease in the number of instructions and operations executed by the vector programs. Programs that execute operations under vector first only present low order data reuses (with distance 1 or 2).

Program	# Insns under Vector First	# Ops under Vector First	% Ops over total	# VF Value (in %)		
				1	2	Other
Swm256	11.06	943.84	11.46	100	0	0
Hydro2d	3.09	312.94	7.65	78.3	21.7	0
Nasa7	0	0	0	0	0	0
Tomcatv	0.81	51.81	6.10	50	50	0
Bdna	0	0	0	0	0	0
Arc2d	0	0	0	0	0	0
Jpeg Decode	0	0	0	0	0	0
Epic	0.03	0.59	2.07	100	0	0
Jpeg Encode	0	0	0	0	0	0
Gsm Encode	0	0	0	0	0	0

Table 3.11 Vector First distribution. Columns 2 and 3 are in millions.

3.7.5 Vector Mask Execution

The Convex C4 vector processor allows the execution of instructions under a calculated mask stored in the **Vector Mask (VM)** register. The VL operations will be carried out, but only those that have the correct value stored in the i^{th} position of the mask will be finally stored in the destination register of the instruction. We have made an analysis of the masks used during the execution of the benchmarks in order to test the effectiveness of masked execution.

Table 3.12 shows the total amount of instructions executed under mask and the percentage of instructions with respect to the total amount of instructions. These data show a relatively small use of the execution under mask in the C4 vector processor. However, taking into account that each vector instruction implies the execution of VL operations, table 3.12 also shows the total amount of operations executed under mask and the percentage of operations referred to the total amount of operations executed. From this table, we can see that the most intensive use of the masked execution is made by the *Jpeg Decode* benchmark with more than 22% of their operations executed under mask. Programs *Jpeg Encode* and *Hydro2d* execute 15.81% and 14.11% of their operations under mask, respectively. The remaining programs execute either very few operations under mask (*Gsm Encode*, *Arc2d*, *Epic*, *Nasa7*, *Swim256* and *Bdna*) or even none at all (*Tomcatv*).

Program	Instructions executed under vector mask		Operations executed under vector mask	
	Total Number ($x10^6$)	% over total instructions	Total Number ($x10^6$)	% over total operations
Swm256	0.003	0.0018	0.264	0.003
Hydro2d	0.569	2.64	576.367	14.11
Nasa7	0.070	0.005	8.056	0.16
Tomcatv	0	0	0	0
Bdna	0.002	0.0005	0.002	0.0002
Arc2d	0.224	0.16	21.214	0.52
Jpeg Decode	0.183	0.35	29.513	22.05
Epic	0.091	0.54	0.145	0.51
Jpeg Encode	1.103	1.21	42.494	15.81
Gsm Encode	0.071	0.007	1.925	1.19

Table 3.12 Instructions and Operations executed under vector mask

The execution of operations under mask can be considered as speculative execution, as all VL operations are carried out but only those that correspond to the right value in the mask are actually used. We can think of the extra operations as misspeculative execution. The analysis of the masks allows measuring the effectiveness of masked execution. All these instructions executed under mask, are properly speculated or not? An operation is speculated "right" if, after the operation has been carried out, the result is actually stored in its destination. All those operations that were carried out but were not stored are misspeculated work. Figure 3.4 shows the distribution of right and wrong speculated operations in the nine programs (recall that program *Tomcatv* is not shown as it does not execute any instruction under mask). Four programs (*Nasa7*, *Bdna*, *Jpeg Decode* and *Jpeg Encode*) have over 50% of right prediction. The other five programs (*Swim256*, *Hydro2d*, *Arc2d*, *Epic* and *Gsm Encode*) have lower values of right speculation, being *Swim256* and *Arc2d* the programs with the worst behaviors (only 7.29% and 7.46% of right speculation, respectively).

Another interesting consideration that we have studied regards the distribution of operations executed under mask among the different instruction types. This study has allowed us to extract the amount of instructions executed under mask for each type of instructions. We have considered six types of instructions: add, mul, div, dyadic, load and store.

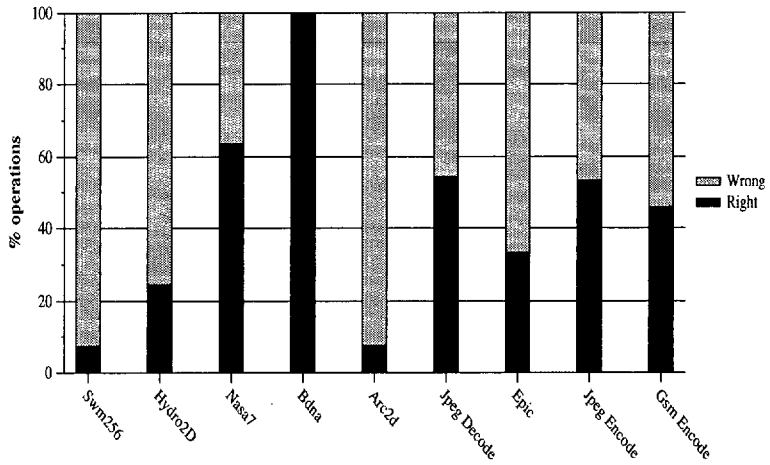


Figure 3.4 Distribution of right and wrong speculative operations in vector programs.

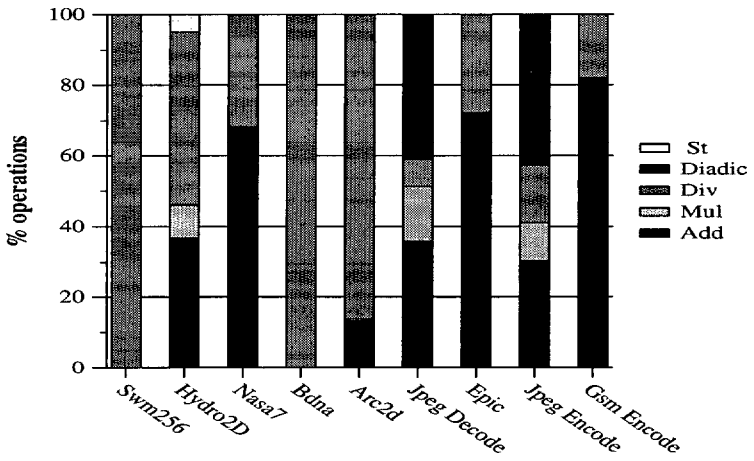


Figure 3.5 Distribution of operations executed under vector mask among the different instruction types.

The first consideration comes from the fact that none of the programs executes load instructions under mask, which is explained because of the use of gather instructions, and it is closely related to the compiler strategy for translating code [SFS00]. Figure 3.5 shows the breakdown of operations executed under mask among the different instruction types. Division and add-like instructions are the most used instructions for execution under mask.

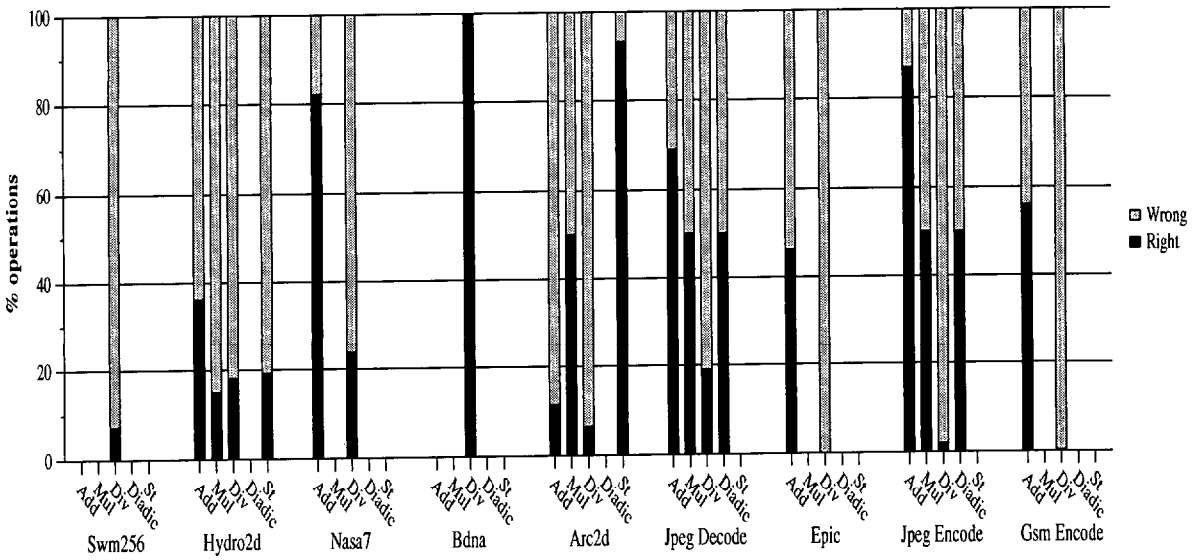


Figure 3.6 Breakdown of right-wrong speculated vector operations.

Finally, we have also studied the effectiveness of execution under mask among the different types of instructions. The results in figure 3.6 show that, in general, there is not a clear correlation between the instruction type and the misspeculation rate. Division instructions are an exception. For divisions, the misspeculation rates are higher than for the rest of instructions in all cases, but program *Bdna*. This result is not unreasonable since division instructions are typically executed in statements such as the following,

$$if\ A(i) <> 0\ then\ B(i) = B(i)/A(i) \tag{3.1}$$

In such a case, misspeculation is determined by the value stored in A(i). In our programs, the A(i) vector is sparsely populated and causes large numbers of misspeculations.

3.8 INFLUENCE OF THE VECTOR LENGTH

As stated in the previous sections, expressing a loop by means of vector instructions reduces the total number of instructions and operations needed for its execution because of the shorter number of iterations needed for the loop execution. A key factor that

influences the number of loop iterations is the maximum vector length that can be expressed in the architecture. The larger the vector length, the shorter the number of loop iterations. However, large vector registers need more chip area to allocate the vector register file, a critical parameter in the design of any microprocessor. Thus, the potential reduction in the number of instructions and operations executed may have a significant cost in terms of chip area.

In this section, we make an study about the behavior of the 10 benchmark programs when varying the maximum vector length from 16 to 32, 64 or 128 elements. The study has been made by measuring the number of instructions and operations executed, and the processor-memory traffic (in terms of load and store operations). These values will be compared with the superscalar results that will be used as reference values of the improvements that can be achieved.

3.8.1 Instructions Executed

As the vector length decreases, the total number of instructions executed increases due to the higher number of loop iterations carried out in each vector loop. Figure 3.7 shows this effect for the ten benchmark programs. This figure shows the total number of instructions executed for different vector lengths, normalized to the number of instructions executed for vector length 128. The figure also shows, as a reference, the total number of instructions executed in the superscalar processor normalized to the number of instructions executed for vector length 128. As already discussed in section 3.4, page 59, the total number of instructions executed is much lower in vector programs than in superscalar programs, and the reason is the semantic gap between the two ISAs. While one scalar instruction carries out just one scalar operation, one vector instruction performs multiple operations over independent data, as many as the vector length register indicates.

Another interesting feature extracted from figure 3.7 is the different sensibility to the variation of the vector length that programs undergo. For example, *Swim256*, *Hydro2d* and *Nasa7* increase a lot the number of instructions executed as vector length decreases, while *Tomcatv*, *Bdna* and multimedia programs undergo a moderate increase in the

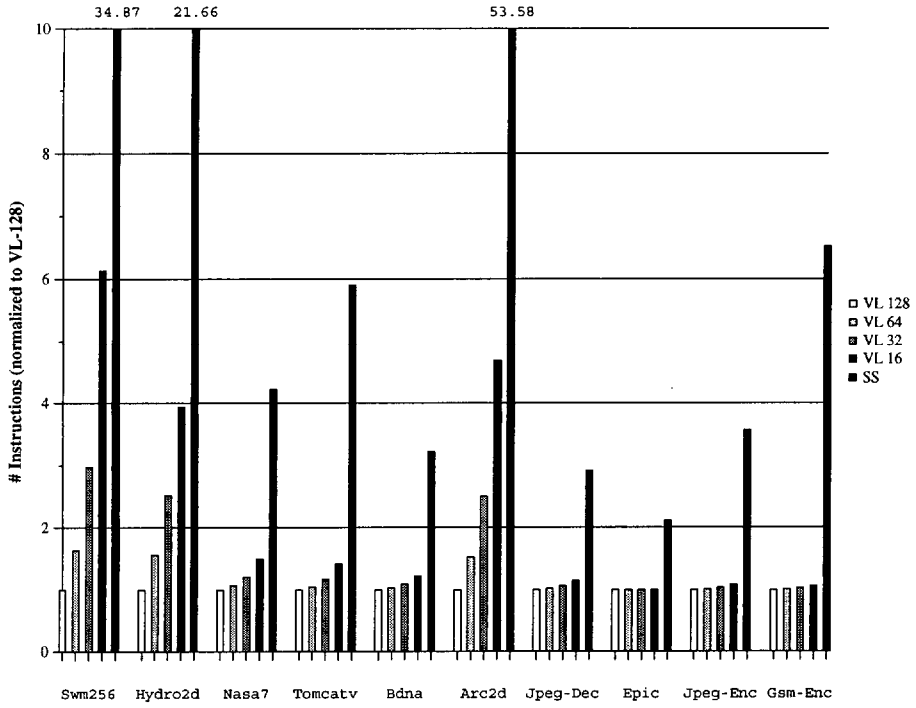


Figure 3.7 Number of instructions executed for the vector processor with vector length 128, 64, 32 and 16 elements, and for the superscalar processor. All these values are normalized to the number of instructions executed for vector length 128.

number of instructions executed. The reason lies on the number and size of the vector loops that each program contains: if there are only a few vector loops with many instructions inside, then we observe a lower increase in the number of instructions executed than if there are many vector loops with few instructions inside. As will be discussed in the following section, these data do not show the real behavior in terms of computational load. The total number of operations executed in these programs is mainly dominated by the vector operations, and this increase in the scalar operations executed is hardly reflected on it.

3.8.2 Operations Executed

In this section, the results about the evolution of the total number of operations executed when the vector length decreases are presented. Figure 3.8 shows the total number of operations executed by the ten programs as the vector length decreases,

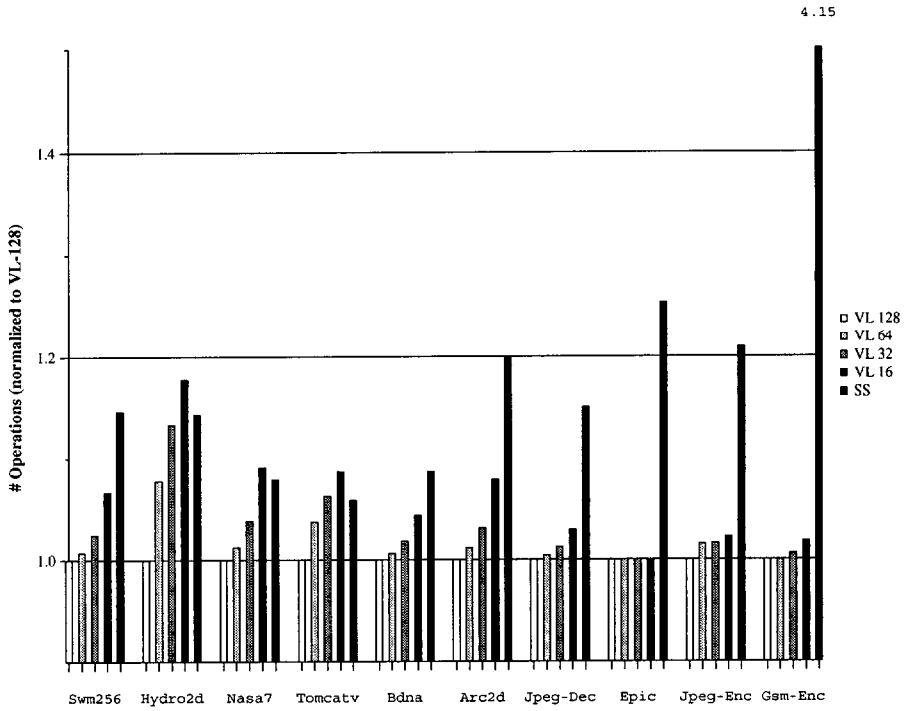


Figure 3.8 Number of operations executed for the vector processor with vector length 128, 64, 32 and 16 elements, and for the superscalar processor. All the values are normalized to the number of operations executed for vector length 128.

as well as the total number of operations executed by the scalar programs. All these values are normalized to the number of operations executed for vector length 128. As expected, the total number of operations increases when the vector length diminishes. In some cases, like in *Swim256*, *Hydro2d*, *Nasa7* and *Arc2d* programs, this increase is clearly observed, while in the rest of the programs the increase is quite slight. The comparison between these results and the superscalar number of operations shows that, even though the total number of operations executed by the vector programs increases, the total number of operations executed by the superscalar programs is far beyond. The exceptions are programs *Hydro2d*, *Nasa7* and *Tomcatv* due to several reasons. On one side, the difference in the quality of the scalar code that both the vector and scalar compilers generate. On the other side, the extra operations executed under vector mask that the superscalar program does not execute. As a measure of the latter reason, recall from section 3.7.5, page 74, and from figure 3.4 that, for example, program *Hydro2d* executes 11% of its operations under mask, from which only 25% are right speculated.

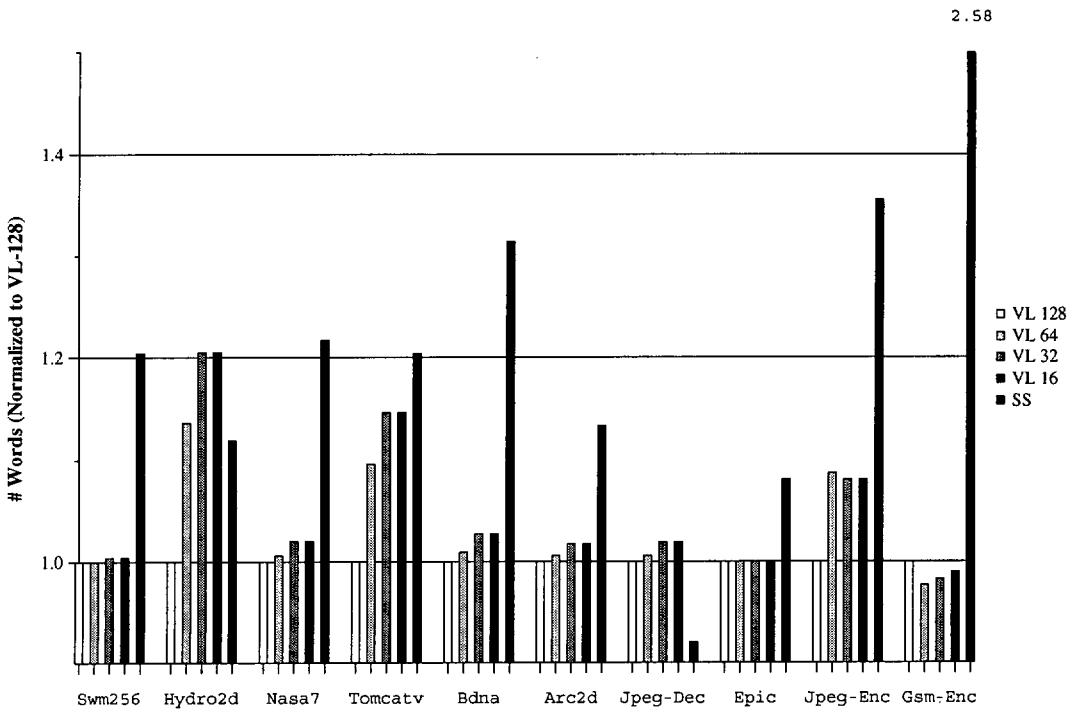


Figure 3.9 Number of words moved between the processor and the memory hierarchy for the vector processor with vector length 128, 64, 32 and 16 elements, and for the superscalar processor. All the values are normalized to the number of words moved for vector length 128.

3.8.3 Processor-Memory Traffic

The study of the total number of instructions and operations executed by the programs, both in the superscalar and vector processors, has a predictable behavior taking into account the semantic characteristics of vector instructions. Another interesting parameter that can be influenced by the vector length is the traffic between the processor and the memory hierarchy, which we have measured as the total number of words moved by load and store instructions. In figure 3.9, we can see that, although there is a variation in the number of memory operations executed, in seven of the ten benchmarks the increase in the memory traffic is negligible. The exceptions are *Hydro2d* and *Tomcatv* programs. In *Hydro2d*, not only the number of memory operations increases as the vector length decreases, but it also exceeds the number of memory operations executed by the superscalar programs. The main increase appears in the number of vector and scalar load operations. As the vector length decreases, the same data must be re-loaded

from memory, thus making the number of load operations increase. In *Tomcatv* happens approximately the same, but to a lower extent. Finally, the comparison between the superscalar and vector behaviors shows that, in general, superscalar programs move much more words with the memory than the vector programs. This is especially true for the 128-element vector registers executions. As the vector length decreases the traffic increases because of the data re-loads, and in one case (the *Hydro2d* program previously commented) the vector traffic overcomes the superscalar traffic. It is worth mentioning the exception of the *Jpeg Decode* behavior, since it is the only program that moves less words with the memory in the superscalar case, no matter the vector length used in the vector programs. The superscalar compiler performs a better code translation, thus yielding a lower memory traffic.

3.9 ANALYSIS BY REGIONS

The analysis of ISA characteristics in complete programs is quite interesting as it allows us to roughly predict the way programs behave during their execution in a superscalar or vector processor. As the aim of this thesis consists in showing that it is worth including a vector unit into a superscalar processor in the field of vectorizable numerical and multimedia codes, let us make a detailed analysis of the ISA characteristics inside S- and D-regions. This study will lead us to a better knowledge of how vector programs behave inside vectorizable pieces of code.

The method for splitting complete programs into S-regions and D-regions has been explained in section 2.6.3, page 48. In this section, we will show the results of the study of the low level characteristics inside S-regions and D-regions. First of all, the evaluation of the importance of S- and D-regions, in terms of number of operations executed, will be presented. Next, the measurements about the number of basic blocks, instructions and operations executed will be shown.

It is important to note that although we will present data about the behavior inside S-regions and D-regions, the most interesting part of this study concerns vectorizable pieces of code, that is, D-regions. The reason is that superscalar and vector compil-

Program	% Ops executed inside D-regions	% Ops executed inside S-regions	Vectorization %	Average size of D-regions	Average size of S-regions
Swim256	99.99	0.01	98.77	380988	21
Hydro2d	99.11	0.89	95.64	44420	396
Nasa7	99.05	0.95	75.62	217266	2077
Tomcatv	83.78	16.22	82.85	1178606	227814
Bdna	88.88	11.12	66.78	82990	10381
Arc2d	99.57	0.43	97.53	135348	585
Jpeg Decode	60.43	36.57	61.00	18146	11877
Epic	62.12	37.88	44.01	5824	3550
Jpeg Encode	78.77	21.23	70.05	1941	523
Gsm Encode	38.90	61.10	36.95	1098	1725

Table 3.13 Percentage of operations executed inside D- and S-regions, vectorization percentage and Average size of D- and S-regions (in operations), extracted from the vector programs. Average size is defined as the total number of operations executed inside S-/D-regions divided into the number of S-/D-regions.

ers should theoretically behave similarly within scalar regions. Although our results contradict this initial intuition, as we have discussed in section 2.7 (see page 49), this mismatch has nothing to do with the superscalar and vector paradigms. The differences come from generational reasons, because the superscalar compiler is much more modern. Moreover, it is our claim that if the vector compiler was built nowadays, using current compiler technology, the scalar parts of vector and superscalar programs would match.

3.9.1 General Characteristics of S-regions and D-regions

Table 3.13 presents the operation breakdown for S-regions and D-regions: percentage of operations executed inside D-regions and S-regions, vectorization percentage (column 5 of table 3.9) and the average size of D-regions and S-regions (in terms of number of operations). Note that the percentage of operations executed inside D-regions is usually larger than the vectorization percentage. This is normal since a D-region *also* contains scalar instructions that correspond to scalar calculations inside loops and/or loop control.

It is worth mentioning that in *Jpeg Decode* program the percentage of operations executed in D-regions is lower than the vectorization percentage because 1.8% of the vector operations are executed outside D-regions due to the overlap between S-regions and D-regions, discussed in section 2.6.3, page 48.

We can see in table 3.13 that *Swim256*, *Hydro2d* and *Arc2d* programs execute more than 99% of their operations inside D-regions, and their vectorization percentage is also over 95%, which means that the number of scalar instructions inside D-regions is small and, therefore, will not affect significantly the performance of D-regions. Therefore, according to the Amdahl's Law, the execution of D-regions is hardly influenced by the low data-parallel execution of scalar code.

Nasa7 presents a similar behavior, executing 99.05% of its operations in D-regions. However, the vectorization percentage is lower (75%), meaning that the number of scalar instructions executed inside D-regions is significant, and their execution will influence the performance of D-regions. *Tomcatv* and *Bdna* have a significant amount of operations in S-regions (around 15%), so the execution of these instructions will influence the global performance. The difference is that, while *Tomcatv* has pure D-regions, *Bdna* has D-regions polluted with scalar instructions, thus limiting *Bdna*'s performance on D-regions.

Multimedia programs execute over 20% of their operations inside S-regions, reaching 61% in *Gsm Encode*, which will strongly determine their global performance and the acceleration that can be achieved by using a vector unit. Again, we can distinguish two types of programs: programs that have pure D-regions (*Gsm Encode* and *Jpeg Decode*), and programs with polluted D-regions (*Epic* and *Jpeg Encode*).

3.9.2 Basic Block Distribution

Table 3.14 presents the total number of basic blocks executed inside S-regions and D-regions, in the vector and superscalar ISAs, for the set of benchmarks. Columns 2 and 5 are the total number of basic blocks executed, recalled from table 3.1. Columns 3 and 4, and columns 6 and 7 contain the number of basic blocks executed inside S-regions

Program	Scalar ISA			Vector ISA		
	Total	S-Regions	D-Regions	Total	S-Regions	D-Regions
Swim256	6.75	0.03	6.72	2.59	0.05	2.54
Hydro2d	263.25	6.82	256.42	11.18	7.14	4.04
Nasa7	24.18	1.06	23.12	16.85	8.13	8.72
Tomcatv	19.91	19.60	0.30	19.95	19.72	0.23
Bdna	33.71	11.60	22.11	41.81	41.25	0.56
Arc2d	25.83	0.32	25.50	4.56	1.57	2.99
Jpeg Decode	8.51	3.17	5.34	6.25	6.22	0.02
Epic	2.00	1.50	0.50	2.92	2.33	0.58
Jpeg Encode	29.69	5.61	24.07	11.67	8.33	3.34
Gsm Encode	28.13	24.50	3.62	15.21	14.50	0.71

Table 3.14 Total number of basic blocks executed in the superscalar machine and in the vector machine; and breakdown into basic blocks executed inside S-regions and D-regions. All columns are in millions.

and D-regions, respectively. The first interesting point from this table is the difference in the number of basic blocks executed inside S-regions for the superscalar and vector platforms; and also inside D-regions for the superscalar and vector platforms. The behaviors are opposite: while inside D-regions all programs execute a larger number of basic blocks in the superscalar platform, inside S-regions the number of basic blocks executed in the superscalar platform is shorter. The behavior inside D-regions follows the same explanation as the total number of basic blocks executed: a shorter number of loop iterations due to the higher semantic content of vector instructions and the predicated execution under vector mask.

However, inside S-regions, the effect is the opposite, and the reason, already discussed in section 2.7, page 49, is that the superscalar compiler is more sophisticated than the vector compiler, as it is more modern and includes additional compilation techniques and optimizations, to better generate and schedule code.

3.9.3 Instruction Breakdown

We will now proceed to analyze the distribution of instructions executed inside the S-regions and D-regions. Table 3.15 presents the number of instructions executed inside S-regions and D-regions for the vector programs. For the S-regions, column 2 contains

the total number of instructions executed, while columns 3 and 4 present the breakdown of instructions in column 2 into scalar and vector instructions. The same measurements are shown for D-regions in columns 5, 6 and 7, in the same table. Several comments can be made regarding this data:

- Looking at columns 2 and 5, we can see that programs present an irregular behavior in terms of the number of instructions executed inside S- and D-regions. While some programs, like *Tomcatv*, *Jpeg Decode*, *Epic*, *Jpeg Encode* and *Gsm Encode*, execute much more instructions inside S-regions, the rest of the programs, i.e. *Swim256*, *Hydro2d*, *Nasa7* and *Bdna*, behave exactly the opposite and execute much more instructions inside D-regions. Given that each vector instruction involves several operations, as many as the vector length, we can not draw any conclusion from this behavior, as it is necessary to know the number of operations that each vector instruction carries out.
- Column 4 in this table partially supports the discussion in section 2.6.3, page 48, about the overlapping effect between S-regions and D-regions. As we can see in this column, and compared to column 3, the fetch and decode units are mainly devoted to the scalar instructions. Vector instructions hardly pollute S-regions, although we must re-visit this claim in next section whenever we present the same measurements in terms of operations executed, instead of instructions.
- Comparing columns 6 and 7, we realize that, even inside D-regions the fetch and decode unit is mainly devoted to the scalar instructions, as D-regions execute much more scalar than vector instructions (from 1.14 to 46 times more scalar instructions for *Tomcatv* and *Bdna*, respectively).

Let us compare these results with the results for the superscalar programs, shown in table 3.16. This table contains the number of instructions executed by the superscalar programs, broken down into instructions executed inside S-regions (column 2) and D-regions (column 3). Contrary to vector programs, superscalar programs behave all the same, that is, all of them execute less instructions inside S-regions. Comparing column 2 in table 3.15 with column 2 in table 3.16, superscalar programs execute less instructions inside S-regions. As mentioned before, the reason is that, as the scalar

Program	S-Regions			D-Regions		
	Total	Scalar	Vector	Total	Scalar	Vector
Swim256	0.47	0.47	0.0001	175.67	100.83	74.84
Hydro2d	36.17	36.16	0.0005	179.53	141.99	37.54
Nasa7	48.57	48.55	0.0180	1262.34	1204.65	57.69
Tomcatv	137.60	137.60	0.0003	14.68	7.85	6.83
Bdna	166.07	166.06	0.0109	338.53	331.35	7.17
Arc2d	9.40	9.34	0.0597	131.30	91.78	39.51
Jpeg Decode	51.50	51.49	0.0125	1.24	0.70	0.54
Epic	10.75	10.75	0.0001	6.14	5.14	0.99
Jpeg Encode	55.71	55.70	0.0109	35.25	24.78	10.46
Gsm Encode	98.43	98.43	0.0037	4.17	3.34	0.83

Table 3.15 Breakdown of instructions executed in S-regions and D-Regions for the vector programs. All columns are in millions.

Program	S-Regions	D-Regions
Swim256	0.27	9438.31
Hydro2d	27.41	4644.34
Nasa7	8.17	5533.86
Tomcatv	117.37	780.29
Bdna	64.59	1564.38
Arc2d	13.56	4892.04
Jpeg Decode	64.05	89.95
Epic	7.75	27.82
Jpeg Encode	51.98	273.03
Gsm Encode	264.15	405.08

Table 3.16 Breakdown of instructions executed in S-regions and D-regions for the superscalar programs. All columns are in millions.

compiler is newer, it is able to generate a higher quality code. The exception are programs *Arc2d*, *Jpeg Decode* and *Gsm Encode*. The reason is that the vector ISA is able to express the same scalar operations using less instructions.

Regarding D-regions, superscalar programs execute much more instructions than vector programs, as observed when comparing column 7 in table 3.15 with column 3 in table 3.16. Let us study a concrete example in the *Gsm Encode* program. In this program, we have made some minor changes in order to get the compiler to vectorize it. The collateral effect of these changes in the source code is the decrease in the total number of instructions executed in the vector program. Although similar changes were tried in

```
for (lambda = 40; lambda ≤ 120; lambda ++) {
  L_result = STEP(0);
  L_result += STEP(1);
  L_result += STEP(2);
  ....
  L_result += STEP(39);
  if (L_result > L_max) {
    Nc = lambda;
    L_max = L_result;
  }
}
```

Figure 3.10 Original scalar loop in the *Gsm Encode* program.

the superscalar program, the superscalar compiler was not able to take advantage of them, and the same effect was not obtained.

For example, figure 3.10 shows an original loop in the program source code. This loop belongs to function `Calculation_of_the_LTP_parameters` that computes the **long term predictor gain** and the **long term predictor lag** for the long term analysis filter. This is done by calculating a maximum of the cross-correlation function between the current sub-segment short term residual signal, which is the output of the short term analysis filter, and the previous reconstructed short term residual signal. In particular, this loop searches for the maximum cross-correlation and coding of the long term predictor lag. The `for` loop calculates each cross-correlation as the cumulative addition of `STEP(i)`, that is,

$$L\_result = \sum_{i=0}^{39} STEP(i),\ being\ STEP(i) = wt[i] \times dp[i - lambda] \qquad (3.2)$$

If this cross-correlation is greater than the maximum (`L_max`), it becomes the new maximum. This is done for all the cross-correlation terms, so that, at the end of the `for` loop execution, the maximum correlation and its position are stored in the `L_max` and `Nc` variables, respectively.

```
lambda_loop:
    ....
    bis $31,6,$20
    subl $20,$14,$1
    addq $1,$1,$1
    addq $1,$13,$1
    bic $1,6,$2
    ldq $2,0($2)              ; load dp[1 - lambda]
    ldl $8,224($30)           ; load wt[1]
    bic $1,1,$1
    extwl $2,$1,$2
    sll $8,48,$1
    sll $2,48,$2
    sra $1,48,$1
    sra $2,48,$2
    mull $1,$2,$1             ; STEP(1) = wt[1] x dp[1 - lambda]
    stl $1,232($30)           ; temporal store
    ...  [up to 40 code blocks like this]
    ldq $8,200($30)           ; reload STEP[i]
    addq $15,$8,$15           ; cumulative addition
    addq $15,$10,$15          ; cumulative addition
    ...  [loads and adds for cumulative addition]
    cmple $15,$8,$1           ; if(L_result <= L_max) => end_if
    bne $1,end_if
    sll $14,48,$1             ; else
    stq $15,168($30)          ; L_max = L_result
    sra $1,48,$1
    stq $1,160($30)           ; Nc = lambda
end_if:
    addl $14,1,$14            ; lambda++
    cmple $14,120,$1          ; if(lambda <= 120)
    bne $1,lambda_loop        ; back to lambda_loop
```

Figure 3.11 Assembler code of the original scalar loop in the *Gsm Encode* program.

The superscalar compiler translates this code into a loop containing 40 pieces of code, each one carrying out one of the 40 calculations of the `STEP(i)` terms, as observed in figure 3.11. Each code block consists of 13 or 14 instructions, depending on where the calculation is stored. If the calculation is kept in a processor register, the final store instruction does not appear. These instructions are distributed as follows:

- 5 instructions to calculate the address of the `dp[i-lambda]` element to be accessed.
- 2 instructions to load the `dp[i-lambda]` and `wt[i]` elements.
- 6 instructions to convert these elements from word to longword.
- 1 instruction to multiply the `dp[i-lambda]` and `wt[i]` elements.
- 1 instruction to store the result in a temporal memory location. If the result is kept in a processor register, this instructions is not needed.

After the `STEP(i)` calculations there is another group of 56 instructions for accumulating them. This group of instructions consists basically of `addq` and `ldq` instructions. Finally, the `if` statement instructions and the loop control instructions close the loop.

Therefore, each loop iteration performs as many as $(24 \times 13) + (16 \times 14) + 56 + 6 + 3 = 601$ instructions. Given that the loop executes 81 iterations, and that the whole loop is executed 7376 times during the program execution, the total number of instructions executed by the superscalar processor for this loop is $601 \times 81 \times 7376 = 359.071.056$ instructions.

Meanwhile, the vector compiler needs some minor changes in the original source code in order to vectorize the loop. Among the different options, we have chosen to vectorize the `lambda for` loop by executing the different iterations in parallel. The first step is to split the `for` loop into two loops, one of them for the cross-correlation calculations, and the other for the maximum calculation. In order to vectorize the first loop, the `L_result` scalar variable is turned into a vector of eighty one scalar elements, as shown in figure 3.12. The eighty one cross-correlation terms are calculated and stored into the L_result vector, in a single go, by using vector instructions with vector length 81.

```
for (lambda = 40; lambda ≤ 120; lambda ++) {
  L_result[lambda] = STEP(0);
  L_result[lambda] += STEP(1);
  L_result[lambda] += STEP(2);
  ....
  L_result[lambda] += STEP(39);
}
for (lambda = 40; lambda ≤ 120; lambda ++) {
  if (L_result[lambda] > L_max) {
    Nc = lambda;
    L_max = L_result;
  }
}
```

Figure 3.12 Vectorized version of the scalar loop in the *Gsm Encode* program.

Therefore, the loop disappears, as well as all the repetitive instructions that must be executed in each loop iteration, like address calculations, index increments, conditional branches, etc. The second `for` loop computes the maximum among the cross-correlation terms. This loop can not be vectorized because of the `if` statement, so that it makes a sequential lookup of the maximum inside the `L_result` vector.

The assembly code generated by the vector compiler is shown in figure 3.13. First of all, the `wt[i]` scalar elements are loaded and converted from half-word into word elements. They are temporarily stored in memory, and will be reloaded later. Then, the `dp[i-lambda]` elements are accessed using a vector load, in a single go. They are also converted from half-word into word elements. Next, the calculation of the `STEP(i)` and its cumulative addition are made by using `axpy` instructions. Finally, the results are moved to memory with a vector store instruction and the second `for` loop looks for the maximum among the elements of the L_result vector. The first loop is codified using 329 linear instructions, that is, no loop is needed as the original loop was vectorized using vector length 81. Some of these instructions are scalar and the rest

```
lambda_loop:
    ld.h -18(fp),s0          ; load wt[0]
    cvth.w s0,s4
    ...
    ld.h -36(fp),s15         ; load wt[39]
    cvth.w s15,s0
    st.w s0,-1228(fp)
    ...
    mov #81,vl               ; vl = 81
    ld.h -2(a5),v0           ; load dp[0 - lambda]
    cvth.w v0,v5
    mul.w v5,s4,v5           ; STEP(0) = wt[0] x dp[0 - lambda]
    ld.h -80(a5),v6          ; load dp[1 - lambda]
    cvth.w v6,v3
    axpy.w v3,v5,v6          ; L_result[lambda]+ = STEP(1)
    .....
    ld.w -1228(fp),s0        ; reload wt[39]
    axpy.w v10,v0,v2         ; L_result[lambda]+ = STEP(39)
    mov #4,vs
    st.w v2,-448(fp)         ; L_result[lambda]

lambda_loop_2:
    ld.w 4(a2),s0            ; L_result[lambda]
    ge.w s4,s0,cc2           ; if (L_result[lambda] > L_max)
    jbr.t end_if,cc2
    ld0.w 4(a2),s4           ; L_max = L_result
    mov.w s3,a8
    add.h a8,#1,a8           ; Nc = lambda
end_if:
    add.w a2,#4,a2
    add.h s3,#1,s3           ; lambda + +
    gt.w a3,a2,cc2           ; if (lambda ≤ 120)
    jbr.t lambda_loop_2,cc2  ; back to lambda_loop_2
```

Figure 3.13 Vector loop at *Gsm Encode* benchmark.

are vector. In particular, there are 192 scalar instructions and 137 vector instructions. Among the vector instructions, 39 instructions are dyadic `axpy` instructions. On the other hand, the second loop executes $81 \times 10 = 810$ scalar instructions. Therefore, the total number of instructions executed by the vector processor is 1139, and the total number of operations executed is $((98 \times 81) + (39 \times 81 \times 2) + 192 + 810) \times 7376 = 112.543.008$ operations.

The comparison shows that, on one hand, the total number of instructions executed by the superscalar processor is huge when compared to the number of instructions executed by the vector processor. On the other hand, the superscalar processor executes three times the total number of operations of the vector processor. Moreover, we can observe that:

- The address computation is made just once in the vector processor while in the superscalar processor it is made 81 times, which implies much more operations executed. Moreover, while the superscalar ISA needs five instructions to calculate this address, the vector ISA only needs two instructions.
- The load of each `wt[i]` element is also made 81 times in the superscalar processor, and just once in the vector processor.
- The `STEP(i)` terms are stored and reloaded later for accumulation in the superscalar processor, while, in the vector processor, the `STEP(i)` terms are calculated and accumulated in the same `axpy` instruction, thus reducing the number of load and store instructions.
- In the vector ISA, the data conversion takes just one `cvth.w` instruction while, in the superscalar ISA, three instructions are needed.

As a conclusion, the reduction in the number of instructions and operations executed is mainly caused by the concise way in which the vector loop is expressed in the vector ISA. This reduction affects not only to the vector computations, but also the typical scalar address calculations or data conversions, as it can be expressed using less instructions.

Program	Scalar Instructions in S-Regions					
	Add	Mul	Ld	St	Control	Total
Swim256	0.18	0.01	0.10	0.09	0.08	0.47
Hydro2d	17.90	0.01	10.61	0.45	7.18	36.16
Nasa7	17.00	1.82	13.51	8.89	7.32	48.55
Tomcatv	104.66	0.01	13.20	0.09	19.63	137.60
Bdna	84.12	3.48	31.28	10.54	36.60	166.06
Arc2d	2.84	0.12	4.14	0.73	1.48	9.34
Jpeg Decode	30.03	2.60	9.48	4.39	4.97	51.49
Epic	4.94	0.36	2.42	0.95	2.07	10.75
Jpeg Encode	25.94	0.01	14.74	7.15	7.84	55.70
Gsm Encode	61.66	5.84	13.82	6.70	10.37	98.43

Table 3.17 Breakdown of scalar instructions for the S-regions of the vector programs. All columns are in millions.

Program	Vector Instructions in S-Regions					
	Add	Mul	Dyadic	Ld	St	Total
Swim256	0	0	0	5	36	41
Hydro2d	0	0	0	5	565	570
Nasa7	1200	1200	3000	4200	8427	18027
Tomcatv	0	0	0	101	206	307
Bdna	0	0	0	5313	5605	10918
Arc2d	0	1068	26100	29772	2791	59731
Jpeg Decode	24	4	0	4732	7751	12511
Epic	6	1	0	9	9	25
Jpeg Encode	10	0	0	5450	5523	10983
Gsm Encode	0	0	0	1858	1858	3716

Table 3.18 Breakdown of vector instructions for the S-regions of the vector programs.

It is important to note that these results about the total number of instructions executed allows us to study the pressure on the fetch and decode units. As superscalar programs execute much more instructions than vector programs, the fetch and decode units in the superscalar processor must be more sophisticated and aggressive to match the performance of the vector processor. These units will consume more chip area in their implementation, and their complexity can make the clock cycle time increase.

Let us now study the instruction distribution inside S-regions using the same categories as in section 3.4, that is, **Add**, **Mul**, **Dyadic**, **Ld**, **St** and **Control** categories. Instructions in vector programs are first classified into scalar and vector, so that the comparison between superscalar and vector programs involves three tables. Tables 3.17,

Program	Instructions in S-Regions					
	Add	Mul	Ld	St	Control	Total
Swim256	0.09	0.006	0.09	0.06	0.02	0.27
Hydro2d	16.90	0.005	3.57	0.18	6.74	27.41
Nasa7	4.64	0.60	0.75	1.15	1.01	8.17
Tomcatv	71.75	0.00	26.01	0.07	19.53	117.37
Bdna	30.30	0.88	18.28	3.67	11.44	64.59
Arc2d	6.30	2.61	3.53	0.17	0.93	13.56
Jpeg Decode	49.47	0.49	8.15	3.20	2.72	64.05
Epic	4.40	0.29	1.18	0.49	1.37	7.75
Jpeg Encode	32.73	0.01	10.33	3.89	5.01	51.98
Gsm Encode	205.15	9.28	22.20	8.79	18.71	264.15

Table 3.19 Breakdown of instructions for the S-regions of the superscalar programs. All columns are in millions.

3.18 and 3.19 contain the number of instructions executed inside S-regions in each category: scalar instructions in vector programs (table 3.17), vector instructions in vector programs (table 3.18) and instructions in superscalar programs (table 3.19).

In general, scalar **Add** instructions are the most frequent inside S-regions. Recalling that the *Arc2d*, *Jpeg Decode* and *Gsm Encode* programs execute more scalar instructions inside S-regions in the superscalar versions, we can see in these tables that the extra scalar operations executed mainly belong to the **Add** class. These programs make an intensive use of data conversions, which are carried out with just one instruction in the vector ISA (`cvth.w`), but this requires several instructions (`extwl`, `sll` and `sra`) in the superscalar ISA.

Regarding vector instructions, they are executed inside S-regions because of the overlapping between adjacent S- and D-regions, as discussed in section 2.6.3, page 48. The number of vector instructions executed inside S-regions is very low, not exceeding in any program 0.63% of the total number of instructions. **Ld** and **St** vector instructions are the most frequent instructions as any vector loop usually starts loading vector data and finishes storing vector results, so when overlapping ocurrs, these "frontier" instructions are executed inside the previous or the following S-region. In the next section, the real computational load in the overlapping zones, in terms of operations, will be studied.

Program	Scalar Instructions in D-Regions					
	Add	Mul	Ld	St	Control	Total
Swim256	84.58	0.01	3.49	4.98	7.74	100.83
Hydro2d	87.18	0.91	21.66	18.93	13.28	141.99
Nasa7	617.43	90.25	298.16	101.27	97.52	1204.65
Tomcatv	5.77	0.19	0.79	0.47	0.60	7.85
Bdna	180.43	45.01	70.36	24.13	11.40	331.35
Arc2d	63.34	0.42	18.40	3.85	5.75	91.78
Jpeg Decode	0.48	0.00	0.13	0.06	0.02	0.70
Epic	2.97	0.00	0.81	0.99	0.36	5.14
Jpeg Encode	19.16	0.00	0.26	2.37	2.98	24.78
Gsm Encode	1.77	0.00	0.72	0.18	0.66	3.34

Table 3.20 Breakdown of scalar instructions inside D-regions for the vector programs. All columns are in millions.

Program	Vector Instructions in D-Regions					
	Add	Mul	Ld	St	Dyadic	Total
Swim256	21.23	11.08	21.26	10.20	11.05	74.84
Hydro2d	12.73	8.14	11.68	4.14	0.82	37.54
Nasa7	10.35	12.51	17.90	7.36	9.55	57.69
Tomcatv	2.34	1.83	1.75	0.74	0.15	6.83
Bdna	1.42	1.93	1.70	1.06	1.04	7.17
Arc2d	6.70	11.36	14.01	4.92	2.50	39.51
Jpeg Decode	0.29	0.08	0.04	0.03	0.08	0.54
Epic	0.49	0.20	0.28	0.01	0.00	0.99
Jpeg Encode	2.33	0.41	4.96	0.34	2.40	10.46
Gsm Encode	0.37	0.03	0.20	0.06	0.14	0.83

Table 3.21 Breakdown of vector instructions inside D-regions for the vector programs. All columns are in millions.

Finally, tables 3.20, 3.21 and 3.22 contain the same instruction distribution across the six categories for D-regions. The more frequent category of scalar instructions inside D-regions is the **Add** category, as can be seen in table 3.20. These scalar instructions carry out all the loop control operations: loop counter increments, condition evaluations, index increments, etc. However, the majority of the computational load of these programs is carried out by the vector instructions, as discussed in section 3.5. The results shown in table 3.21 do not allow drawing any conclusion about the instruction distribution, as different programs behave differently, and the real computational load comes from studying the number of operations executed rather than the number of instructions executed. For the scalar programs, data presented in table 3.22 show

Program	Instructions in D-Regions					
	Add	Mul	Ld	St	Control	Total
Swim256	3890.19	1750.01	2565.32	1167.97	64.80	9438.31
Hydro2d	1865.15	631.73	1466.25	398.10	23.08	4644.34
Nasa7	1695.16	982.21	2092.82	644.94	118.70	5533.86
Tomcatv	275.32	162.67	248.46	88.51	5.31	780.29
Bdna	579.07	361.17	432.31	169.60	22.21	1564.38
Arc2d	1678.10	1069.92	1521.06	543.70	79.25	4892.04
Jpeg Decode	74.02	0.00	8.14	4.54	3.24	89.95
Epic	14.39	2.78	8.41	0.25	1.97	27.82
Jpeg Encode	146.32	18.35	78.09	11.09	19.17	273.03
Gsm Encode	302.68	24.29	60.83	12.93	4.34	405.08

Table 3.22 Breakdown of instructions inside D-regions for the superscalar programs. All columns are in millions.

that the **Add**-type instructions are the most frequent instructions, except for *Nasa7* program, followed by the **Ld**-type instructions.

3.9.4 Operation Distribution

Although the instruction study provides us with some important information that allows predicting the behavior of the pressure in the fetch and decode units, measurements about the operation distribution inside S-regions and D-regions show the real computational load of the programs. It is important to note that data referring to scalar operations will match data already presented about scalar instructions, as one scalar instruction performs exactly one operation. For that reason, tables referring to scalar operations will not be repeated but will be recalled from previous sections instead.

The first measurements, shown in table 3.23, refer to the total number of operations executed inside S-regions and D-regions. Columns 2 and 5 in this table present the total number of operations executed inside S-regions and D-regions, respectively. Columns 3 and 4, are the data in column 2 distributed among scalar and vector operations, for S-regions. Columns 6 and 7 present the same operation distribution for D-regions. The analysis of these data shows several points:

Program	S-Regions			D-Regions		
	Total	Scalar	Vector	Total	Scalar	Vector
Swim256	0.47	0.47	0.0005	8230.50	100.83	8129.67
Hydro2d	36.16	36.16	0.0053	4049.55	141.99	3907.56
Nasa7	48.69	48.55	0.1496	5091.20	1204.65	3886.55
Tomcatv	137.60	137.60	0.0028	710.70	7.85	702.85
Bdna	167.37	166.06	1.3192	1329.93	331.35	998.58
Arc2d	17.61	9.34	8.2757	4071.69	91.78	3979.91
Jpeg Decode	52.95	51.49	1.4698	80.88	0.70	80.18
Epic	10.75	10.75	0.0021	17.63	5.14	12.49
Jpeg Encode	57.06	55.70	1.3680	211.70	24.78	186.92
Gsm Encode	98.62	98.43	0.1989	62.78	3.34	59.44

Table 3.23 Breakdown of operations executed in S-regions and D-Regions for the vector programs. All columns are in millions.

- The majority of the operations are executed inside D-regions, which was expected to some extent as the set of programs is quite vectorizable (recall comments in sections 3.7 and 3.9.1, pages 67 and 83, respectively).
- The total number of vector operations executed inside S-regions due to the overlapping effect is very low, reaching the worst case in program *Jpeg Decode* with 1% of the total number of operations executed. The execution of these vector instructions outside D-regions hardly pollutes S-regions. The exception is program *Arc2d*. For this program, the total number of vector operations executed inside S-Regions is very low (only 0.2% of total number of operations executed). However, the fact that the size of the S-regions is also short, makes the overlapping operations become an important part of them, reaching 47% of the total number of operations executed inside these regions. These data provide quantitative support to section 2.6.3, page 48, where the importance of the overlapping between consecutive S- and D-regions was discussed.
- Inside D-regions, the majority of the operations executed are carried out by means of vector instructions, which was already shown in a different way in table 3.13, page 83.

Comparing this data to the superscalar programs requires recalling table 3.16 in the previous section, while considering table 3.23. It is not worth comparing results inside

Program	Vector Operations in S-regions					
	Add	Mul	Dyadic	Ld	St	Total
Swim256	0	0	0	45	461	506
Hydro2d	0	0	0	39	5267	5306
Nasa7	7200	7200	36000	25200	74047	149647
Tomcatv	0	0	0	908	1958	2866
Bdna	0	0	0	656968	662305	1319273
Arc2d	0	102884	4993800	2888027	291061	8275772
Jpeg Decode	744	124	0	591205	877904	1469853
Epic	192	32	0	984	984	2160
Jpeg Encode	294	0	0	682770	685029	1368093
Gsm Encode	0	0	0	99482	99482	198964

Table 3.24 Breakdown of vector operations inside S-regions for the vector programs, in absolute number.

S-regions in these two tables since the behaviors inside S-regions in terms of operations and instructions mainly matches. The reason is that the only difference between the numbers of operations and instructions comes from the vector operations executed inside the S-regions which hardly influence, as discussed above.

Comparison inside D-regions shows that superscalar programs execute much more operations inside D-regions than vector programs (from 8% to 57% more operations, being *Gsm Encode* program a special case that executes 646% more operations due to the data vectorizations already commented). The notion of a region being executed using a superscalar or a vector processor involves the idea of a computational unit that can be expressed and executed in two different ways. From that point of view, these data quantitatively support the idea that the vector ISA is able to express and execute a vectorizable computational unit using less operations and instructions.

We now turn to the analysis of the operation distribution inside regions. Table 3.24 contains the vector operation distribution inside S-regions. As pointed out above, these vector operations are executed inside S-regions because of the overlapping between consecutive S- and D-regions. The most frequent operations are in the **Ld** and **St** categories, being program *Arc2d* a special case that executes more that 50% of these overlapping operations in the **Dyadic** category.

Program	Vector Operations in D-regions					
	Add	Mul	Dyadic	Ld	St	Total
Swim256	1891.00	946.09	2200.43	2061.73	1030.41	8129.67
Hydro2d	1294.86	826.84	168.21	1199.57	418.05	3907.56
Nasa7	716.26	683.89	659.50	1273.50	553.38	3886.55
Tomcatv	214.52	175.47	26.01	205.44	81.39	702.85
Bdna	169.75	233.50	258.31	207.25	129.76	998.58
Arc2d	631.41	1075.73	478.63	1326.26	467.87	3979.91
Jpeg Decode	37.82	10.59	21.18	6.04	4.53	80.18
Epic	5.45	2.65	0.00	4.29	0.08	12.49
Jpeg Encode	51.69	15.57	63.94	48.85	6.85	186.92
Gsm Encode	16.08	0.84	23.59	15.26	3.65	59.44

Table 3.25 Breakdown of vector operations inside D-regions for the vector programs. All columns are in millions.

Finally, table 3.25 contains the vector operation distribution inside D-regions for the vector programs. Since most of the total number of operations in a program are executed as vector operations inside D-regions, this table is very similar to table 3.7, in which the total operation distribution for vector programs was shown, and the same comments can be applied.

3.10 HYBRID BENCHMARKS FOR VECTOR EXECUTION: CHARACTERISTICS

As stated in section 2.7, page 49, the evaluation of the ILP+DLP architecture will be carried out by using hybrid vector programs. The hybrid version of each program is built by merging the D-regions generated with the vector compiler and the S-regions generated by the superscalar compiler. Therefore, our hybrid vector benchmarks are a quite good approximation to real programs compiled using an hypothetical modern vectorizing compiler.

In the following sections we will present the characteristics of the hybrid vector benchmarks in terms of instructions and operations executed as well as their vectorization characteristics.

Program	S-regions	D-regions	Total
Swim256	0.27	175.67	175.94
Hydro2d	27.41	179.53	206.94
Nasa7	8.17	1262.34	1270.51
Tomcatv	117.37	14.68	132.05
Bdna	64.59	338.53	403.12
Arc2d	13.56	131.30	144.86
Jpeg Decode	64.05	1.24	65.29
Epic	7.75	6.14	13.89
Jpeg Encode	51.98	35.25	87.23
Gsm Encode	264.15	4.17	268.32

Table 3.26 Number of instructions in the hybrid set of benchmarks: inside S-regions, inside D-regions, and total number of instructions. All columns are in millions.

3.10.1 Instruction Breakdown

Table 3.26 contains the total number of instructions executed by the hybrid vector benchmark programs. Each program executes as many instructions inside S-regions as the S-regions of the scalar program (column 2), and as many instructions inside D-regions as the D-regions of the vector program (column 3). Finally, column 4 shows the total number of instructions executed. Provided that the hybrid vector programs have a lower number of instructions executed inside S-regions, the general trend is the decrease in the total number of instructions executed. However, three programs (*Arc2d*, *Jpeg Decode*, and *Gsm Encode*) behave oppositely. As commented in section 3.9.3, page 85, the reason is that the vector compiler generates a better code inside S-regions, as these programs make data conversions that are expressed in the vector ISA by using a lower number of instructions.

3.10.2 Operation Distribution

Although the total number of instructions executed by the benchmark programs provides interesting information about the ISA, the total number of operations executed by the programs is the real measure of their computational load. As shown in table 3.27, and comparing it with column 3 in table 3.6, the hybrid vector programs execute, in general, a lower number of operations than the vector programs. The exceptions are

Program	S-regions	D-regions	Total
Swim256	0.27	8230.50	8230.77
Hydro2d	27.41	4049.55	4076.96
Nasa7	8.17	5091.20	5099.37
Tomcatv	117.37	710.70	828.07
Bdna	64.59	1329.93	1394.52
Arc2d	13.56	4071.69	4085.25
Jpeg Decode	64.05	80.88	144.93
Epic	7.75	17.63	25.38
Jpeg Encode	51.98	211.70	263.68
Gsm Encode	264.15	62.78	326.93

Table 3.27 Number of operations in the hybrid set of benchmarks: inside S-regions, inside D-regions, and total number of operations. All columns are in millions.

Program	# instructions		# vector operations	Vect %	Avg. VL
	Scalar	Vector			
Swm256	101.01	74.84	8129.68	98.77	109
Hydro2d	169.40	37.54	3907.56	95.84	104
Nasa7	1212.82	57.71	3886.70	76.21	67
Tomcatv	125.22	6.83	702.85	84.87	103
Bdna	395.94	7.19	999.90	71.60	139
Arc2d	105.34	39.57	3988.18	97.42	101
Jpeg Decode	64.75	0.55	81.65	55.32	147
Epic	12.89	0.99	12.49	49.21	13
Jpeg Encode	76.76	10.47	188.29	70.88	18
Gsm Encode	267.49	0.83	59.64	18.18	71

Table 3.28 Basic operations counts for the hybrid vector benchmarks on the vector machine (Columns 2-4 are in millions).

Jpeg Decode and *Gsm Encode* programs. These programs execute a larger number of operations than the vector programs due to the reasons discussed in the previous section. The rest of the hybrid vector programs execute many fewer operations than the superscalar programs, as observed when comparing the total number of operations in table 3.27 with column 2 in table 3.6.

3.10.3 Vector Characteristics

Provided that the hybrid programs have improved S-regions, leaving the D-regions unchanged, their vector characteristics hardly differ from the vector characteristics of

the pure vector programs. Table 3.28 shows, for the hybrid vector programs, the more relevant vector characteristics, that is, the vectorization percentage and the average vector length. As commented in section 3.7, page 67, the vectorization percentage is defined as the ratio between the number of vector operations and the total number of operations executed by the program. Given that the total number of operations has changed, the vectorization percentage will also suffer some variations from that presented in table 3.9. However, this difference is very slight in almost all programs, as can be seen comparing columns 4 from tables 3.28 and 3.9. The differences reach values between 0.2% and 5.2% in all programs, except for *Arc2d*, *Jpeg Decode* and *Gsm Encode*. In these programs the vectorization percentage diminishes in 0.11%, 4.78% and 18.77%, respectively.

Of course, the variation in the vectorization percentage depends on the variation undergone in the total number of operations executed. For this reason, program *Swim256*, for example, hardly changes its vectorization percentage, as the reduction in the number of operations hardly influences the 8230.5 million operations executed inside D-regions. On the other side, in program *Gsm Encode*, this variation is rather significant due to the large increase in the number of scalar operations executed in the hybrid vector program.

Table 3.28 also shows the average vector length for the hybrid vector programs. Recall from section 3.7 that the average vector length is the ratio between the number of vector operations and the number of vector instructions. Since the D-regions have been kept unchanged, both, the number of vector instructions and operations, remain constant in the hybrid programs, so that the average vector length does not change from that presented in table 3.9.

The rest of the vector characteristics, that is, vector length distribution, vector first execution and vector mask execution, presented in section 3.7, do not change as they are measured inside D-regions, which remain the same in hybrid vector programs.

3.11 SUMMARY

In this chapter we have presented a study about the superscalar and vector programs from the instruction set architecture level. We have started discussing the benefits of the vector ISA in terms of instruction fetch bandwidth, memory system performance (both latency and bandwidth) and datapath control.

Next, we have studied the distribution of basic blocks, instructions and operations executed. Vector programs execute fewer basic blocks, instructions and operations than superscalar programs due to the higher semantic content of vector instructions and the predicated execution under mask. Each vector instruction carries out several operations on independent data, which reduces the total number of loop iterations, and consequently, it also reduces the total number of operations for loop counter increments, index increments, condition evaluations and conditional branches. Therefore, less aggressive fetch and decode units are needed, which can yield a faster clocking of the datapath.

The study of the data types used by the programs show that while numerical programs mainly work with 64-bit data types, multimedia programs work with 32-bit data. Moreover, multimedia programs carry out an important part of their operations with 8-bit and 16-bit data.

The vector characterization of the ten benchmark programs analyzes the vector length and vector stride distributions, and the vector first and vector mask executions. The vector length distribution shows the way in which the different programs use the vector registers. As the size of the vector registers decreases, more vector instructions with the larger vector length are executed. The vector stride distribution shows that, although the stride-1 memory accesses are the most frequently used, there is an important part of memory accesses that are carried out with other larger strides. The vector first execution analyzes a particular feature of C4 ISA [Con93]. This characteristic avoids reloading data when executing code with data reuses inside, thus reducing the total number of load and store operations. The execution under mask consists in executing all the vector operations specified in one vector instruction, but only storing the results

according to the bits of a vector mask that has been previously calculated. In this way, the number of conditional branches and if-then-else structures are reduced, thus making the programs less control-dependent. The operations that have been carried out, but whose result have not been stored, can be seen as misspeculated operations. The study of the masks has shown that, in spite of the large amount of misspeculated work, it is probably worth using the mask execution because of the reduction in the number of control instructions.

Another important topic discussed in this chapter is the influence of the vector length on the total number of instructions and operations executed, as well as in the traffic between the processor and the memory system. As the vector length increases, the total number of instructions and operations decreases, since a lower number of loop iterations is needed. Opposite to this effect is the increase in the size of the vector register file as the vector length increases, which means that a greater chip area is needed to allocate the vector register file inside the processor chip.

The analysis of the number of basic blocks, instructions and operations executed inside S-regions and D-regions allows us to make a more detailed study of the pure scalar and vector ISAs. The superscalar programs execute many more basic blocks, instructions and operations inside D-regions than the vector programs. The analysis of the S-regions shows that the quality of the scalar code generated by the superscalar compiler is higher. The solution consists in building a new set of benchmark programs starting from the pure superscalar and vector programs. Each hybrid vector program consists in the original S-regions from the superscalar version plus the original D-regions from the vector version. The hybrid vector programs execute fewer instructions and operations than the pure vector programs while they keep almost the same vector characteristics.

In summary, we might state that, given the benefits of the vector ISA, it is worth exploring the possibility of including a vector functional unit in a current superscalar architecture. Next chapters will be devoted to the analysis of this proposal.

4

A SUPERSCALAR PROCESSOR WITH A VECTOR UNIT

Summary

In this chapter we describe in detail our proposed architecture, focusing first on the datapath design, the vector register file and the vector unit included, and then describing the memory hierarchy design. We also introduce a new cache design, named "vector cache" that is aimed at delivering small vectors to the vector registers through a wide path.

4.1 INTRODUCTION

Our ILP+DLP proposal consists in adding vector facilities to a current superscalar processor by including vector instructions in current scalar ISAs, and adding a vector register file connected to the functional units already present in the superscalar processor.

The critical factor that determines the overall performance of vector architectures has always been the memory system. Traditional parallel vector supercomputers have been backed with a memory system consisting of multiple banks and sections that can deliver the high bandwidths that the processor requests. Moreover, to achieve low latencies, high performance SRAM memory banks are typically used. This type of memory systems is very expensive due to several factors. First, SRAM chips, used to achieve low latency, are very expensive and offer a modest capacity. Therefore, a large number of chips are needed to build a large memory system. Second, the interleaving required to provide high bandwidth requires many independent sections and large crossbars, which consume a large number of chips and board interconnections.

Designing a memory system that the superscalar market can afford has become a primary objective of this work. Moreover, we think that the use of alternative memory devices with better performance/cost ratios, such as current RDRAM parts instead of expensive SRAM ones is mandatory. In our proposed architecture we introduce a memory system based on a novel cache design called "vector cache" which is a specially designed memory that can deliver full cache lines to the processor through a wide connection.

The rest of this chapter is devoted to the description of the proposed architecture: the datapath and the memory system, including the vector cache.

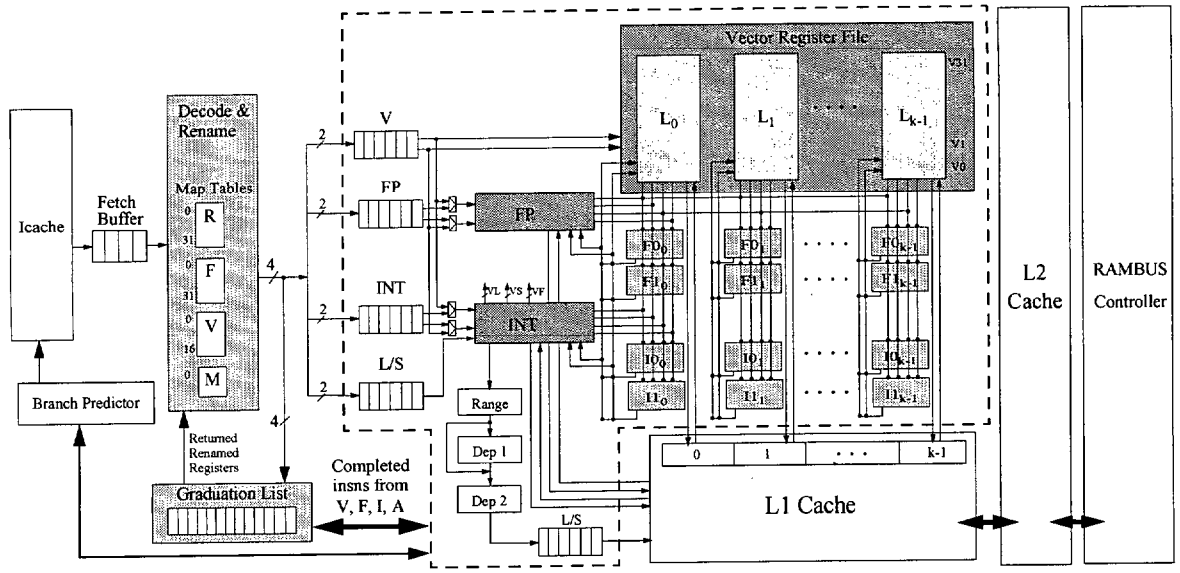


Figure 4.1 Modeled architecture.

4.2 GENERAL DATAPATH

In this section, we describe the components of the datapath of our proposed architecture. We will see the different pipelines present in the architecture, the units that have been added in order to obtain vector functionality and the general characteristics of the datapath operation.

Figure 4.1 shows the main components of the datapath for the proposed architecture. The same figure is reproduced at full page at the end of the chapter, page 129, for a more detailed analysis. Essentially, the architecture is modeled after a MIPS R10000 processor [Yag96], with the addition of a vector register file connected to both the integer and the floating point functional units.

The general operation of the pipelines closely follows the R10k. Instructions are fetched and sent to the decode stage in order, where they are renamed. Whenever a new instruction enters the decode stage, a slot in the reorder buffer is allocated. Instructions enter and exit the reorder buffer in program order. The renaming step consists in translating each virtual register into a physical register by using a mapping table.

There are four independent rename tables, one for integer registers, one for floating point registers, one for vector registers and one for mask registers. Each table keeps its own list of free registers. When an instruction defines a new value of a logical register, the entry in the mapping table for that logical register must be updated with a new physical register. The physical register is taken from the appropriate free list and the mapping table is updated with that register number. Moreover, the old value in the mapping table entry is stored in the reorder buffer slot of that instruction. It is important to note that the information kept in the reorder buffer slots refers to register numbers, and not to register values, which means that only a few bits are needed to keep this information, so that the reorder buffer is a small size structure. Finally, the old physical register is returned to the free list whenever the instruction is committed.

Whenever branch instructions enter the decode/rename stage, the processor predicts the outcome of every branch and speculatively executes the branch based on this prediction. The branch predictor is a gshare predictor [McF93] implemented similarly to SimpleScalar Toolset [BA97].

After the renaming stage, instructions are sent to one of the four issue queues according to the instruction type. Instructions wait in these queues until their operands are ready and then arbitrate for a free functional unit. The processor keeps decoded instructions in four instruction queues, which dynamically issue instructions to the execution units. The queues allow the processor to fetch and decode instructions at its maximum rate without stalling because of instruction conflicts or dependencies. Instructions in every queue can be executed out-of-order. The processor dynamically issues an instruction as soon as the functional unit is ready to accept it, as long as the instruction does not depend on other instructions that have not completed.

In general, instructions are fetched, decoded, issued and graduated in their original program order, but they may be executed and completed out of program order.

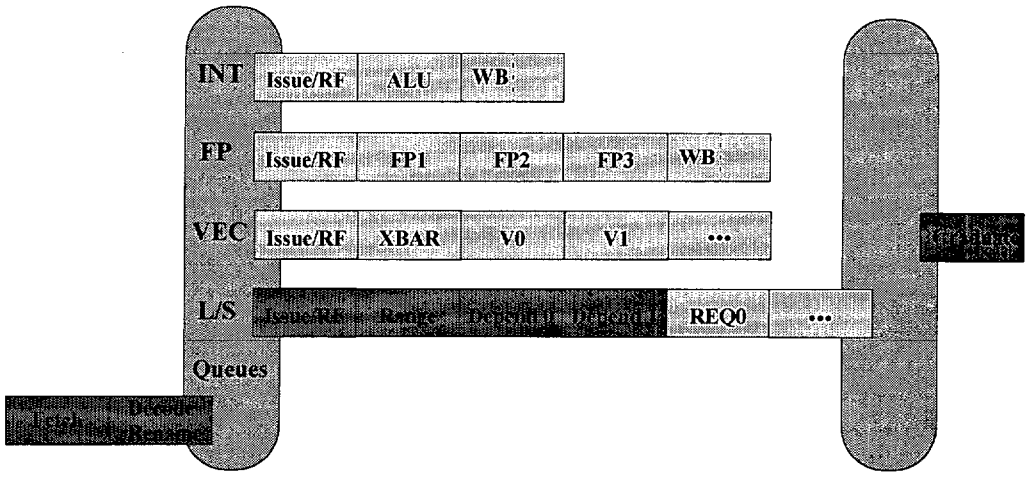


Figure 4.2 The four types of pipelines in the proposed architecture. The stages that are carried out in order are shown in dark grey color.

Instruction Pipelines

There are four types of pipelines in the proposed architecture, one for each type of instruction, as can be observed in figure 4.2.

After fetch and decode/rename stages, instructions are sent to one of the queues according to their type. The superscalar core effectively fetches and decodes several instructions in each cycle. Each decoded instruction is included in one of the four instruction queues, and each queue is able to issue instructions to one of the four types of pipelines:

- The integer queue issues instructions to the integer pipeline.
- The floating point queue issues instructions to the floating point pipeline.
- The load/store queue issues instructions to the load/store unit.
- The vector queue issues instructions to the vector pipeline.

The integer, floating point and vector queues check the status of all instructions in the queue slots until they become ready, and only at that time they are issued to the

appropriate functional unit for execution. Instructions in the Load/Store queue work in a slightly different way. The first four stages of the Load/Store pipeline are carried out in-order, and the rest of the stages can be performed out-of-order, based on dependence information computed in previous stages and operand availability. The first four stages of the load/store pipeline perform the following tasks:

Issue/Rf Stage: It issues the instruction and it reads the register file in order to obtain the right values of the instruction operands. In particular, the integer register file is read in order to obtain the value of the base memory register that allows calculating the memory address to access. If necessary, the vector length and vector strides registers are also read.

Range Stage: It performs the address range calculation, defined as the range of all the addresses that the memory instruction can potentially modify. The range is defined as all the memory addresses falling between the Range Start and the Range End, that is,

$$Range\_Start \leq address \leq Range\_End$$

$$base\_address \leq address \leq base\_address + (VL - 1) \times VS,$$

being VL the vector length register and VS the vector stride register. Note that, in this expression, the multiplier could increase the processor cycle time. However, this does not happen because of two reasons. On one side, the $(VL - 1)$ term is short. As the maximum vector length is 128, it will not be greater than 7 bits. On the other side, the product $(VL - 1) \times VS$ can be kept in an internal register that will be implicitly updated whenever VL or VS registers are modified.

Dependence Stages: They compute the dependences between the current instruction and all previous instructions in the queue, using the information provided by the range stage. Scalar memory instructions can bypass the second dependence stage as their dependence calculations are easier to perform. Once an instruction does not have any dependence it can proceed to the next stage.

Once the instructions have finished these stages, they can issue the memory requests out-of-order.

Regardless the instruction type, whenever one instruction finishes its execution is marked as completed. However, as instructions are committed in program order, each completed instruction can not be graduated until all previous instructions have been graduated. An exception are store instructions, which are only allowed to be executed and update memory when they are the head of the reorder buffer, that is, when they are the oldest uncommitted instructions. This commit model enables a straightforward implementation of precise exceptions, which in turn allows supporting virtual memory.

The Vector Register File and additional registers

The major addition to the superscalar processor core is the vector register file (VRF), together with its connections to the available functional units, and a set of special purpose registers: the vector length, vector stride, vector first and vector mask registers.

Three major parameters must be chosen concerning the Vector Register File: (1) number of logical vector registers, (2) number of physical vector registers for renaming, and (3) length of each individual vector register.

We expect that a reasonable ISA will provide at least 16 programmer visible vector registers, to give enough flexibility to the compiler to schedule vectorized code. Previous studies [EVS97] show that the number of physical registers required to support renaming must be at least twice as many as the number of logical registers. Consequently, we settled on 32 physical vector registers. On the other hand, the length of each vector register has to be chosen carefully. Traditional vector supercomputers have always chosen vector lengths as large as possible (128, 256 or more). However, given the area constraints of a microprocessor, we can not make our vector registers arbitrarily large. Moreover, our studies show that not all multimedia applications can take advantage of vector lengths larger than 8 or 16 elements [QCEV99] [QCEV01].

For the purpose of this thesis, we have chosen to study different vector lengths: from 16 up to 128 elements. A vector length of 16 elements implies that the full vector register file will use 4 Kbytes of storage ($32\ registers \times 16\ elements \times 8\ bytes$). We consider that this size is a reasonable choice for current and near-future technology. However, in the

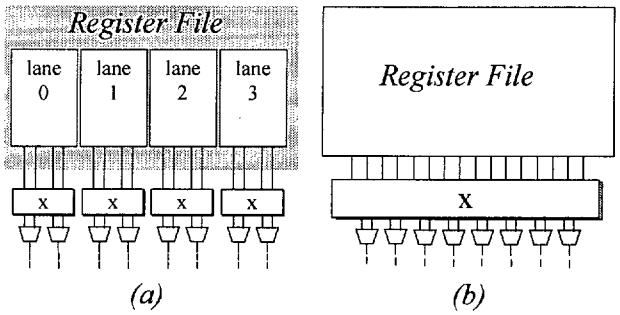


Figure 4.3 Example of (a) Parallel lanes in vector computing, (b) Superscalar Replication of functional units.

mid-term, as integration increases, larger vector registers might fit on-chip very well. Therefore, we have also studied a vector register file where each vector register has 32, 64 or 128 elements (requiring a total of 8, 16 and 32 Kbytes of storage, respectively).

The vector instructions can operate both on integer and floating point data. Therefore, the functional units of the original machine are also shared by the vector unit. In the simplest design, (shown in figure 4.1) the vector unit can have a maximum of two vector instructions in progress. As we scale up the architecture, we add *parallel lanes* to all functional units. For example, in figure 4.1, k parallel lanes are shown. The addition of parallel lanes can be achieved with a relatively simple logic by replicating the functional units, splitting each vector register across each lane, and assigning each functional unit to a certain lane, as shown in figures 4.1 and 4.3. The different elements of a vector register are interleaved across lanes, allowing all lanes to work independently of each other. In contrast, figure 4.3 also shows how this kind of scaling would be implemented in a traditional superscalar design. Note the sharp difference in the complexity of each organization's crossbars. In the superscalar style, adding one functional unit requires adding at least two read ports to the register file, which directly increases the crossbar complexity from $N \times M$ to $(N + 2) \times (M + 2)$. However, in the vector case, adding two functional units to this example only requires splitting the VRF into five lanes, distributing the vector elements across the lanes and adding a 4×4 crossbar that connects the new lane to the new functional units. Of course, in hardware terms the

Register File	Logical registers	Physical registers	Read ports	Write ports
INT	32+3	64	9	3
FP	32	64	5	3
Vector	16	32	5 (per lane) $k \times 5$	3 (per lane) $k \times 3$
Mask	1	18	3	2
CC	6	12	1	1

Table 4.1 Register files characteristics.Three registers have been added to the integer register file, which correspond to VL, VS and VF. The number of read and write ports is global to each register file, except in the vector register file where the number of read and write ports per lane are also specified.

addition of a small crossbar is much more simpler than the extension of an already large one.

Besides the vector register file, a set of special purpose registers have been added. These are the Vector Length (VL), Vector Stride (VS), Vector First (VF) and Vector Mask (VM) registers, which have already been introduced in chapter 3. The vector length and vector stride registers control the execution of all vector instructions while the vector first and vector mask registers are only present in those instructions where they are explicitly specified. The vector length, vector stride and vector first registers are handled in the same way as integer registers, so that they are located and renamed in the integer register file. Renaming is necessary because vector instructions are also executed out-of-order, and each instruction needs its particular value of VL and VS. Moreover, as instructions involving the assignment of new values to VL, VS and VF imply the use of integer registers, it is a natural way to locate these special purpose registers in the integer register file.

The vector mask register has as many bits inside as the maximum vector length. In the largest case the vector mask register is 128-bit long, being larger than any integer register. It is also necessary to rename this register in order to provide any vector instruction with the right mask value. In figure 4.1 it must be noticed that the vector mask registers must be accessible to the functional units in order to execute operations under vector mask. Table 4.1 summarizes all register files present in the proposed architecture. In this table we can also observe the characteristics of the Condition

Code register file (CC), which contains six logical and twelve physical condition code registers.

4.3 THE MEMORY HIERARCHY

In this section we describe the design of the memory hierarchy that provides the superscalar+vector architecture with all the data that it needs. We will introduce the vector cache, a specially designed cache that is able to deliver "small vectors" to the processor. We will analyze the advantages of including data caches into a scalar architecture with vector functional units and we will explain why they were not used in the old vector architectures. We will also address the bandwidth problem, and we will propose an easy but effective method to increase the effective bandwidth of the accesses already described in past works [RBS96] [CMMP95]. Finally, we will explain the memory hierarchy designs that we will evaluate in our proposal in the following chapters.

4.3.1 The Vector Cache: A Cache for Vector Accesses

First of all, we discuss the design of a cache-based memory system tuned to our vector unit. The goals of our cache design are: first, to provide high bandwidth to the vector register file. Second, to allow this bandwidth to scale up as we increase the number of functional units. Third, due to the very distinct natures of the scalar memory stream and the vector memory stream, to minimize the conflicts between them. Fourth, to guarantee that the processor cycle time is not in jeopardy due to the inclusion of a high bandwidth port to the vector register file.

The bandwidth problem

The cache hierarchy approach has been the solution for superscalar architectures to overcome the ever increasing speed gap between processor and main memory. Current superscalar processors integrate at least two levels of cache hierarchy. Typically, we can find a small and fast on-chip L1 cache, backed up by a large (1–4MB) off-chip L2 cache.

However, advances in logic integration have allowed recent superscalar processors (such as the Alpha 21364 [Ban98]) to include both caches on-chip.

As the number of instructions executed in parallel increases, data caches with higher bandwidth will be required. This is specially true in the case of vectorizable codes, characterized by their memory hungry nature. To obtain high bandwidth from a cache, several requirements must be satisfied. First, multiple TLB translations must be made in parallel. Second, the cache must provide multiple access ports. For a small number of ports (say 2 or 3) this is usually done by implementing a true multi-ported cache (either time multiplexed as in the Alpha 21264 [Kes99] or with multiple cache copies as in the Alpha 21164 [BK95]). However, research results show that for large number of ports, say 4 to 16, this is not feasible and alternative designs using multiple banks or hybrids of multi-bank and multi-port must be used [JNT97] [RTAD97].

Thus, no obvious solution seems to be available for scaling current superscalar processors up to issuing four or more memory accesses per cycle. Even though accesses tend to exhibit high spatial locality, the out-of-order nature of the references makes it difficult to take advantage of this locality in order to maximize the effective rate of data accesses. In many cases, the spatial locality often translates into bank collisions in multi-bank data caches, thus decreasing the effective bandwidth of the cache ports.

Contrary to scalar memory traffic, vector memory accesses offer an easy way to exploit the spatial locality of references. A vector access is characterized by three different parameters: initial address, vector length (that is, number of references of the vector access), and vector stride (that is, the distance between consecutive references inside the vector access).

If the vector access stride is 1, then we have maximum spatial locality and an opportunity to take advantage of the bandwidth available on chip. Considering the density of current chip technology, why not having a very wide path from the cache into the vector registers ? When a 16-element stride-1 vector request is launched, it is easy to envision the first level cache delivering full cache lines to the processor as opposed to delivering a single element per cycle. Current superscalars can not take advantage of

this, because each cache access is independent of all other cache accesses. Meanwhile, a stride-1 vector access typically needs a single TLB translation[1] and then it accesses multiple elements from the cache.

A cache for vector accesses ?

Traditional vector architectures have not used data caches for vector data. This design decision was based on three different claims [HS93] [KSF+94], that we question as follows:

- Claim 1: *Memory latencies are amortized among the large number of data referenced in a vector access.* While being true that vector machines are characterized for their ability to tolerate long memory latencies better than superscalar architectures, it has been shown in [EVS97] that typical vector codes are actually very sensitive to increases in memory latency, resulting in severe losses of performance. Therefore, on-chip data caches may help to reduce the average access time of the vector memory accesses, thus improving performance.
- Claim 2: *Vector workloads are claimed to have low spatial and temporal locality, and it is already exploited by the large vector registers.* As we will see later, vectorized versions of typical superscalar codes still exhibit enough spatial and temporal locality to be exploited by a data cache, even with large vector registers. Furthermore, a data cache is an ideal solution to overcome the problem of the vector spill code, typically found when we have a limited number of registers at the ISA level.
- Claim 3: *Caches do not have enough bandwidth to satisfy the bandwidth requirements of a vector processor.* The cache bandwidth mismatch problem can be solved by exploiting the intrinsic spatial locality of vector accesses and, indeed, this is what our vector cache design proposes. Vector instructions allow exploiting spatial locality of numerical and multimedia codes because the VL is implicit in the instruction semantic, which supplies a natural prefetching mechanism. Moreover, the vector approach orthogonally rearranges the memory references in comparison to the way

[1] When crossing page boundaries two translations would be made

the conventional approach does, thus explicitly exposing the spatial locality with no interleaving between heterogeneous memory access streams.

For all these reasons we consider that the inclusion of a cache for vector accesses is a promising alternative that is worth exploring for our proposed architecture. The following sections make an in depth description of the vector cache datapath, as well as they describe the way the vector cache is included in the memory system, that is, the memory hierarchy.

The vector cache

We have designed a high bandwidth data cache targeted at accessing stride-1 vector accesses by loading a whole cache line instead of individually loading the vector elements (see figure 4.4). Then, a shift and mask logic correctly aligns the data, eliminating residuals. The vector cache is able to achieve high bandwidth ratios for stride-1 accesses, even for unaligned addresses. The two main hot spots of this design, which will be deeply discussed in the following sections, are the alignment logic and the impact of strided vector accesses in the overall performance.

This cache design heavily borrows from the ideas introduced in [CMMP95] [RBS96] where a similar cache architecture was proposed to deal with the problem of unaligned accesses in instruction caches. In these previous papers, however, only the load path to the cache was considered, since the cache was used only to fetch instructions. Our design, in contrast, must include a store path to write data into the cache.

Figure 4.4 shows the proposed design. The cache is two-bank interleaved so that two consecutive cache lines can be accessed. Therefore, a whole vector access overlapped over two different lines can be accessed simultaneously. This scheme requires three different logic blocks to perform the data alignment: *an interchange switch*, since we may need to swap the position of the two lines, *a shift*, to align the lines accessed to the specified initial address, and a logic to *mask* the unused data based on the vector length of the required vector access.

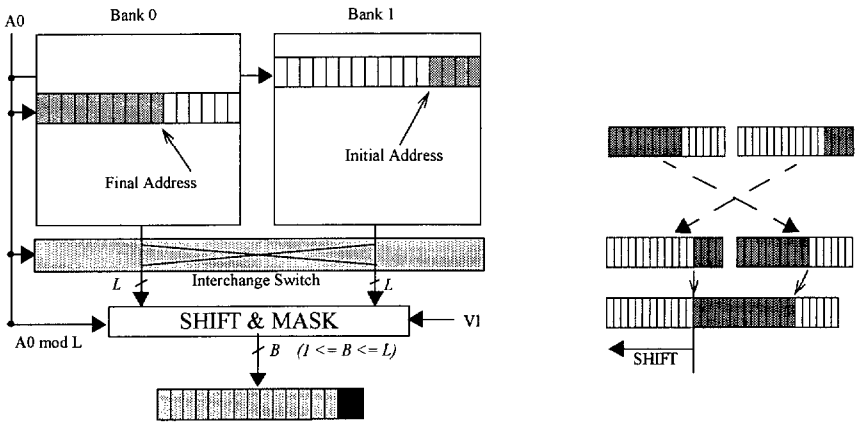


Figure 4.4 The dual bank structure of the vector cache.

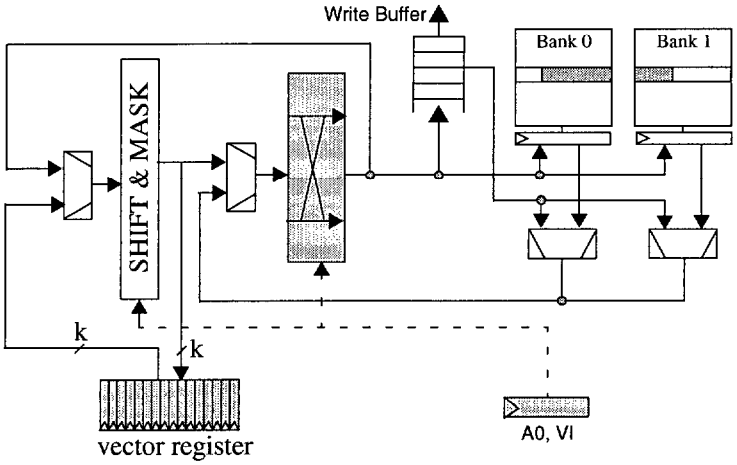


Figure 4.5 Load/Store paths for the vector cache.

Figure 4.5 shows the corresponding datapath that allows loading and storing from the vector cache. For load operations, the cache lines loaded from both banks are first swapped conveniently, and then shifted and masked. Once the vector has been accessed, it is written into the appropriate vector register using a wide bus that writes all vector elements in parallel. In [RBS96] it is claimed that a load could effectively be completed in a single cycle, because: (1) the left-shift amount (A_0 modulo line size) is known at the beginning of the cache access cycle, and has the entire cache access to fanout to the shifter datapath, and (2) assuming a transmission gate barrel shifter, data pass through only one transmission gate delay.

Unfortunately, this is not possible for store operations. First, we need to shift and mask the data from the vector register, and then swap the partition conveniently. Once these operations have been carried out, the store over the cache banks can be effectively performed. Therefore, we will assume that the required time to perform a store operation is two cycles.

Unaligned accesses

An important question to take into account is why designing such a complex alignment network for the vector accesses instead of forcing all vector loads and stores to use naturally aligned addresses?

Three main reasons have led us to model the shift and mask logic. First of all, our benchmark programs are compiled on a machine with no alignment constraints (a Convex C4) and, therefore, many unaligned accesses occur during program execution. Second, the analysis of multimedia workloads shows that misaligned accesses are the norm and not the exception. Multimedia and DSP codes make a great use of small matrices that are difficult to align. Third, even for numerical codes, where padding to force the proper alignment is somewhat easier, we feel that the ability to access misaligned vectors provides the compiler with much more freedom. Moreover, as we shall see in the following chapter, our final design uses the vector cache previously described as the Level-2 cache (not as an L1). Therefore, pipelining this shift and mask logic should have minimal impact on processor cycle time and, at worst, it would only add a single cycle of latency to the vector memory path.

The write buffer

The write buffer for our vector cache has been designed following the results presented in [SC97]. We have implemented a coalescing write buffer with a width equal to the cache line size, since it has to be able to insert both scalar and vector accesses. The retirement order is FIFO, as usual, except for those cases where a load hits a write buffer entry. In those cases, we have chosen a flush item only policy combined with a data bypass mechanism to reduce the latency of the access. The normal retirement is

produced when the number of write buffer entries is greater or equal than X (retire-at-X policy), where X is half the depth of the write buffer. Additionally, our model does not allow merging stores into an entry that is being retired, although they can update other buffer entries while a retirement takes place. The write buffer that we have modeled has a single port. Therefore, in the event of a cache line miss, two cycles are required to test if the desired data are located in the write buffer (since a vector access may be overlapped over two different write buffer entries).

The non-blocking mechanism

Intuitively, supporting bandwidth hungry codes such as numerical and multimedia applications requires a significant degree of non-blocking operation in the cache.

We have designed a Miss Status Holding Register (MSHR) table [CB92] specially targeted at our proposed vector cache. When the maximum vector length is 16 a single vector access may cause two cache line misses, since we assume that the maximum vector length equals the size of a cache line and we also allow misaligned accesses. Therefore, each entry of the MSHR needs to hold miss state information for two cache lines. Scalar accesses and vector accesses with strides larger than the cache line size, only make effective use of a half of each MSHR entry. Similarly, when the maximum vector length is 128 a vector access may cause eight or nine cache line misses. In this case we need four or five MSHR entries available in order to keep the non-blocking mechanism in the cache.

When a line arrives from the upper level of the memory hierarchy, it clears the "pending line" bit in the appropriate MSHR entries. An entry will only be freed when none of its two line descriptors are asserted. In case of having more than one entry to be retired, the retirement is given the highest priority based on the age of every entry.

Another important feature is the possibility of making what we have named "data compression" in the vector memory accesses. The vector cache is targeted at providing a given amount of 64-bit words, so that when the processor issues a stride one vector memory instruction of a smaller data type, say for example 32-bit words, the double

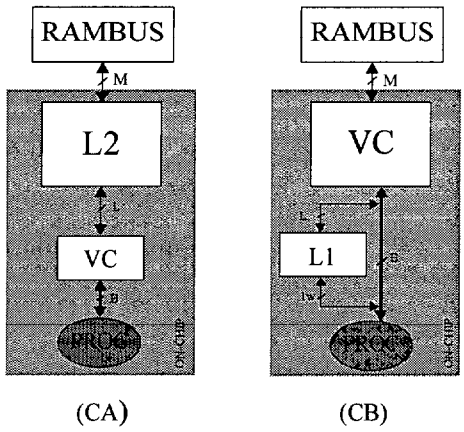


Figure 4.6 The vector cache data path.

of elements can be provided in one single cache access, thus providing all the data in a shorter time, and needing a half number of MSHR entries.

4.3.2 Cache Hierarchy

The design of the cache hierarchy scheme is a critical key for the performance of the whole system. After describing the basic principles of the "vector cache", now it is time to decide whether this vector cache should act as a first level or as a higher level cache. Our goal is the design of a scheme which would become the best trade-off between performance on numerical/multimedia applications and typical non-numerical and/or non-vectorizable codes. Furthermore, critical issues such as the inclusion property between the cache levels, and, what is more important, the coherence of the accesses must be taken into account since they will have a great impact on the final performance. We have developed two alternative cache hierarchies that we will describe now and will be evaluated and compared in the following chapter.

Model A: Vector Cache in L1

The most naive approach would simply make the vector cache a first level cache and provide a wide path between this cache and the vector registers. This first design,

which we refer to as the **CA** cache model, is shown in figure 4.6. Both, vector and scalar accesses, are sent to the L1 cache using a $B\times$ 64-bit words wide bus. A stride-1 vector access of VL elements that hits in the cache uses the full bus bandwidth and takes just $\lceil VL/B \rceil$ cycles to complete. Scalar accesses and non-stride-1 vector accesses, however, can not take advantage of the full width of the bus and simply load a maximum of one word per cycle. Finally, the L2 data cache, which is assumed to be on-chip, is a conventional cache that will be connected by a bidirectional bus to an external RAMBUS controller [Cri97]. Since we would like our model to be extensible to multiprocessing, our simulators faithfully model all the coherency traffic required to maintain the *inclusion* property. The data contained in L1 must be a strict subset of the data contained in L2. For instance, if a 128-byte line is evicted from L2, then care must be taken to invalidate, if present, all its four 32-byte sub-lines in L1.

There are several disadvantages to having the vector cache at the first level:

- First, the complexity of the vector cache could jeopardize the processor cycle time. The processor cycle time is mainly defined by the time taken to access the L1 data cache. As the vector cache has extra stages in order to switch, shift and mask the data, processor cycle time could be increased by the time needed to make these tasks, which would involve an important performance loss.
- It would be difficult to extend the vector cache design to a multiported configuration. In order to provide the processor with extra ports to the memory system, the vector cache must be re-designed as these extra ports also require their own switch, shift and mask logic, as well as temporal registers, multiplexors, connections to the write buffer and other logic elements.
- Storing to the vector cache takes two cycles, which could potentially compromise scalar performance. In current superscalar architectures, both, load and store, take one cycle to execute. Therefore, making scalar stores to take a double number of cycles would introduce additional execution cycles, thus yielding a serious performance loss. However, vector stores are not so affected by this 2-cycle latency because, as previously discussed, vector instructions are latency tolerant. The two-

cycle latency is amortized among all the vector elements, providing a very small mean latency.

- The vector working set is expected to be several times larger than the capacity of the L1 cache although, in many cases, it might fit in a large L2 cache. If this is the case, then constantly missing in L1 to find data in L2 incurs control overheads that could be cut down if a direct path to L2 were provided.

Model B: Vector Cache in L2

All these problems can be overcome in the **CB** model, also shown in figure 4.6. The CB model has a conventional single-ported L1 data cache with small lines, for example 32 bytes. Scalar accesses are sent to the L1 conventional data cache at a maximum rate of one element per cycle. On the other hand, vector accesses bypass the L1 conventional cache to access directly the L2 vector cache [Hsu94] [Sha99]. These accesses are performed at a maximum rate of B elements when the stride is one, and at a rate of 1 element per cycle for any other stride. The L1 data cache uses the wide path for its refills. Of course, the problem when bypassing the L1 is data coherency, an issue that we consider in the following subsection.

The CB model effectively "decouples" the scalar working set from the vector working set. Scalar accesses are made exactly in the same way as in a pure superscalar processor, with latency also remaining unchanged. Vector accesses pay a slightly larger latency but, in return, they hit much more often in the L2. Since vector code is more latency tolerant than scalar code, this increase in latency should have a minimal impact on performance, while the better hit rate should provide the vector unit with a very large effective bandwidth.

Coherency Protocol in the Memory System

When a vector access is made to the L2 bypassing the L1, two integrity problems must be taken into account. First, as was the case in CA, we must follow the *inclusion* principle. Second, data coherency between L1 and L2 must be maintained. Following the inclusion principle implies that, whenever an L2 cache line is replaced, "invalidation"

commands are sent to the L1 to flush the appropriate lines. Observing the inclusion principle allows us to design a very simple coherence protocol to solve the data coherency problem. Every cache line (in L1 and L2) has an *Exclusive* bit associated to it. A certain cache line can be in exclusive state either in the L1 or in the L2 caches, but never in both caches simultaneously. The idea is that a scalar access to the L1 or a vector access to the L2 can only use the data in the line being accessed if its exclusive bit is set. If the bit is not set, we have two different situations.

If a scalar access finds a line without the exclusive bit, it means that the line has been read or written by the vector unit using the bypass path. Therefore, the L1 requests the line to the L2 and acquires the exclusive state.

If a vector access finds a line without the exclusive bit, it means that, potentially, a part of the line might have been written in the L1 by a scalar access (this could be a true hazard or a false sharing situation). The L2 sends a 'flush' command to the L1 to get the most recent copy of the line (acquiring the exclusive state) and then serves the access.

Although other protocols could be more efficient, they have the drawback of being more complex and time consuming. We have chosen this protocol because of its simplicity and acceptable results.

4.3.3 RAMBUS Main Memory

The *Direct Rambus DRAM* (RDRAM) [Cri97] is the memory technology that provides high bandwidth, which is the characteristic that the vector execution needs due to the high bandwidth that is demanded to the main memory system.

We have modeled a 128MB *Rambus* main memory system, shown in figure 4.7, which contains a *RDRAM* controller driving 8 *Rambus* chips and 16 memory banks per *Rambus* chip (128 banks total). This structure is replicated to obtain the desired 64-bit word at 400Mhz, since the CPU is assumed to work at this frequency. The bus connecting

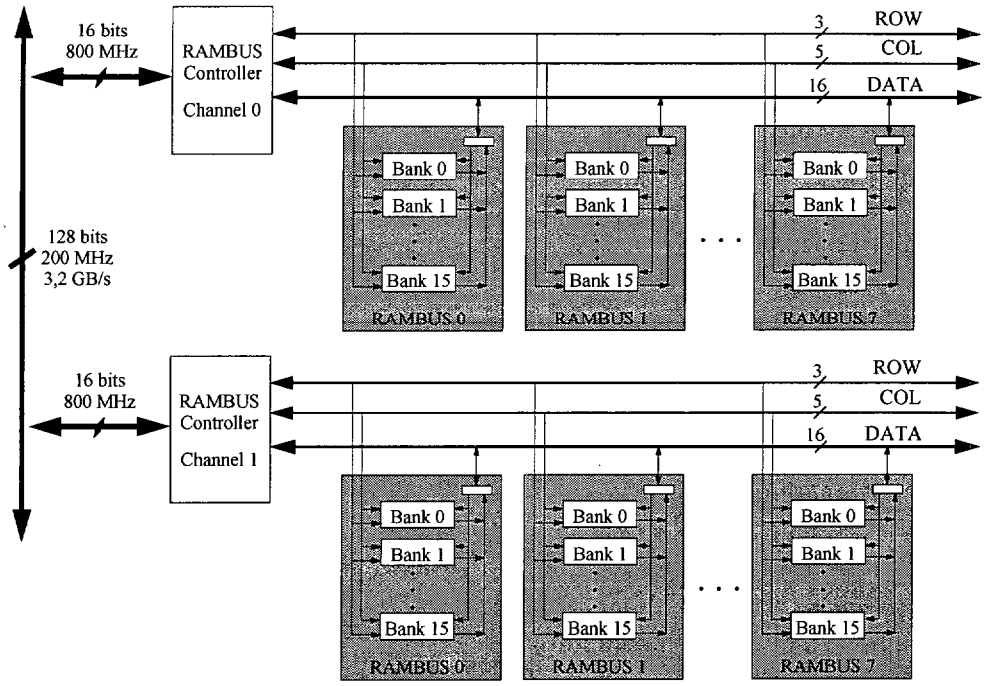


Figure 4.7 The main memory system modeled using RDRAM technology.

the L2 to the Rambus is a 128-bit wide, bi-directional data bus running at 200Mhz, resulting in a memory bandwidth of approximately 3,2 Gb/s.

The L2 cache controller sends cache line requests to the *Rambus* controller, which manages the *RDRAM* modules based on the *RDRAM* protocol. Two optimizations have been added to the Rambus controller: first, no *ROW* command is generated on a bank hit, and, second, the re-organization of queued requests is allowed in order to maximize throughput. Note that a store request from the cache will have to wait for any previous load operation to avoid collisions in the main bus.

4.4 SUMMARY

In this chapter we have presented the superscalar+vector architecture. We have described the datapath, which is very similar to a current superscalar datapath. The main difference comes from the addition of a Vector Register File (VRF) and its con-

nections to the functional units present in the architecture. Some other special purpose registers have been added: the vector length, vector stride, vector first and vector mask registers. The datapath uses out-of-order execution and register renaming. The general operation of the pipelines closely follows the R10k. Instructions are fetched and sent to the decode stage where they are renamed. There are four rename tables, one for integer registers, one for floating point registers, one for vector registers and one for mask registers. Once renamed, instructions go to the appropriate queue where they wait until their operands are ready and then arbitrate for a free functional unit. Instructions in every queue can be executed out-of-order, but they are graduated in program order. There are four types of pipelines in the datapath: integer, floating point, vector and load/store pipelines. While the three first pipelines simply monitor the status of the instructions in their queues and when they become ready they are issued to the appropriate functional unit, the latter pipeline works differently. It has four initial stages that are carried out in order: Issue/RF, Range, Dependence1 and Dependence2. These stages are aimed at computing dependences among instructions, so that the rest of the stages can proceed out-of-order, based on the dependence information calculated before.

On the other side, the memory system is based on a new cache design, called "vector cache", which is able to deliver small vectors to the processor through a wide path. Although traditional vector processors have not used caches for vector accesses, current technology seems to indicate that this decision must be revisited. The vector cache is a dual banked memory from which full lines are read. These lines are switched, shifted and masked as necessary, and then data are sent to the processor. In this way, a high bandwidth, low latency data path to the memory is achieved. Among the different ways in which the vector cache can be included in the memory hierarchy, we explore two options: either include it as the L1 cache, or include it as the L2 data cache adding a direct path from the vector cache to the processor. Both models are backed with a *Rambus* main memory. These two models will be evaluated in the following chapters.

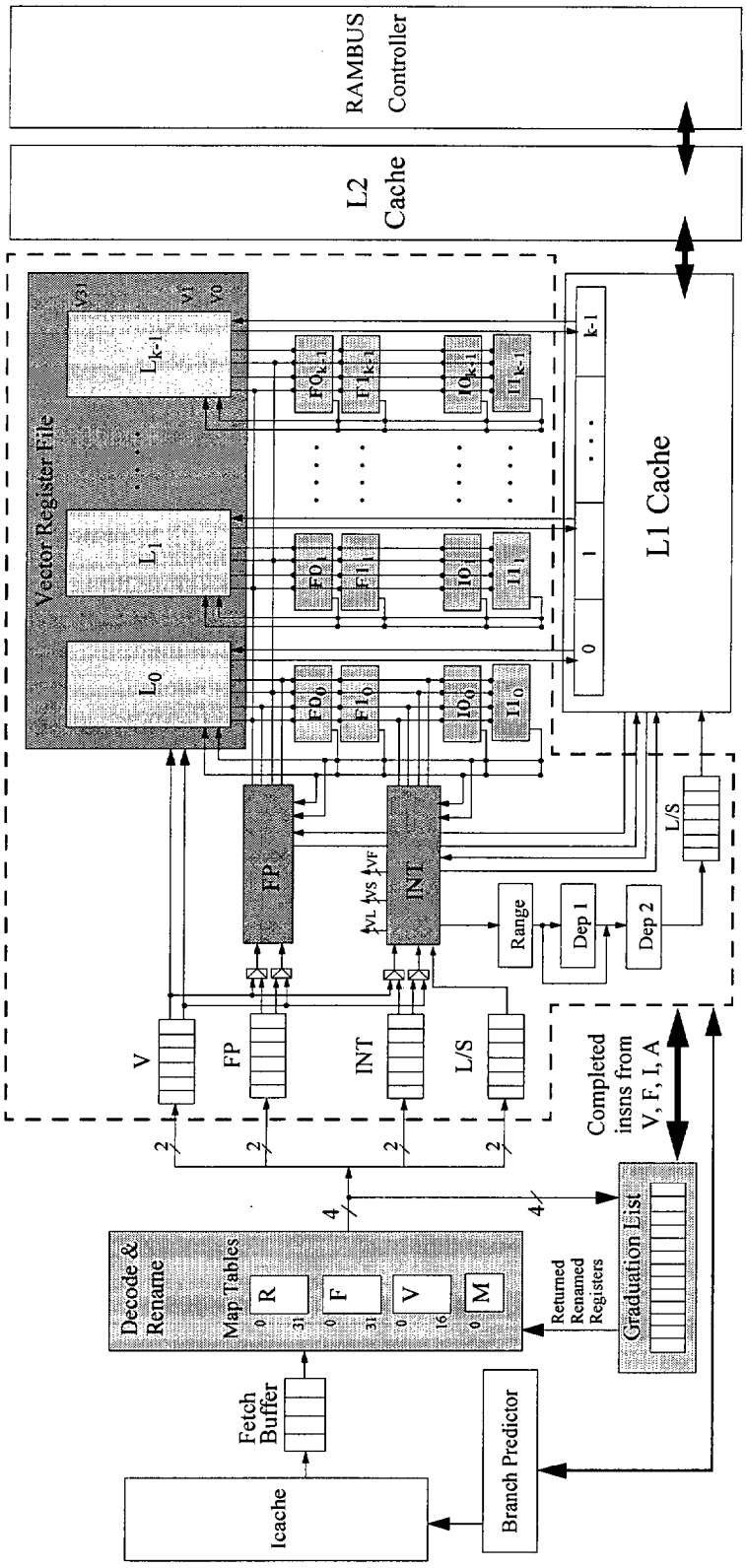


Figure 4.8 Modeled architecture.

5

POTENTIAL PERFORMANCE AND SCALABILITY

Summary

In this chapter we will present some performance results of the proposed superscalar architecture with a vector unit and we will compare it with a traditional superscalar core. We are interested in two different aspects: performance scalability and real performance. The performance scalability is studied assuming a perfect memory system. Afterwards, a study of the real performance is carried out considering the two different cache organizations introduced in the previous chapter.

5.1 INTRODUCTION

After describing the ILP+DLP architecture and the memory hierarchy, this chapter evaluates their performance. We present some performance results of the superscalar architecture with a vector unit (SSV architecture, from now on) that we proposed, and we compare it with a traditional superscalar core. The performance evaluation is carried out in two different aspects:

- On one hand, we study the scalability of the proposed architecture in terms of the amount of instruction/operation level parallelism achieved with a perfect memory. We wish to understand how performance scales as we add more and more functional units to the processor.
- On the other hand, we will present some results about the performance of the proposed architecture backed with our cache hierarchy, discussing the relative merits of the CA and CB cache organizations, and comparing it to a typical superscalar core with a traditional memory hierarchy.

Scalability study with a perfect memory hierarchy

The study of scalability and potential performance analyzes how the SSV architecture behaves as the memory and computational power increases. Intuitively, the greater the number of functional units, the higher the performance. In the limit case, there could be as many functional units as the vector length, so that a vector instruction could be carried out in just one cycle. The new contribution comes from the way the functional units are added: we keep the same two functional units and we increase the unit width by replicating them, but controlled with the same lines.

This first approximation will be carried out considering a perfect memory system. The goal of this study is to know the maximum potential performance that the SSV architecture might reach. To this end, we will start by presenting performance figures assuming a perfect cache system with infinite bandwidth and 100% hit rate.

This chapter will also investigate the effects of vector length on scalability. The maximum vector length will influence the potential performance that the SSV architecture can achieve. Therefore, we will carry out the study for maximum vector lengths of 16 and 128 elements, which will give us an upper and lower bound of performance for any other vector lengths comprised between these two.

Apart from studying the overall performance, we will also discuss measures inside D-regions and S-regions, so we will be able to understand what the contributions of D-regions and S-regions to overall performance are, and how these contributions limit overall performance.

For comparison purposes we have chosen a typical superscalar architecture (SS architecture, from now on) as a reference, and we have compared the relative performance improvements of both architectures as they are scaled up.

Scalability study with a real memory hierarchy

The second important study of this chapter refers to the achieved performance of the SSV architecture when a real memory hierarchy is attached to the processor. Obviously, adding a real memory system will cause a significant performance loss because of the additional cycles that every memory access will take to execute. The main factors that influence the quantitative importance of the performance loss are the following:

- **The characteristics of the memory system.** In vectorizable programs, the real bandwidth that the memory system can provide is especially important. This real bandwidth depends on the configuration and design of the memory system. As introduced in chapter 4, we propose two memory systems.

  In this chapter, we study both memory models, not only from the performance point of view, but also in terms of the amount of traffic that both models generate through the memory, as well as in terms of the hit/miss rate that the processor perceives whenever it carries out a memory access. Starting from these data, we will discuss the relative merits of both cache organizations.

Regarding the election of the best memory model, we will see that different factors, such that the size of the working set or the vector stride, will provide different performance behaviors in our set of benchmarks. Moreover, in order to make a decision about which is the best of both memory models, it is also important to consider what the behavior of low vectorizable programs would be, when executed in an architecture that has been tuned to execute vectorizable programs.

- **The stride of the vector memory accesses.** In both models the spatial locality property is exploited by fetching sequential data from the outer memory level. However, as mentioned before, performance will vary depending on the vector stride and the vector length of the memory accesses. When the vector stride of the memory accesses is 1, all data brought from memory are actually used. However, for vector strides greater than 1, not all transferred elements are really used. The greater the stride, the higher the number of elements of the cache lines that are not used, thus meaning that the effective memory bandwidth that the processor perceives is smaller.

- **The vector length of the vector memory accesses.** The effect of the vector length is, in some way, related to the previous discussion. A vector memory access of 128 elements with stride 1 will take less number of cycles if the maximum vector length is 128 than if it is 16. The reason is not only that less instructions must be fetched, decoded, issued, executed and graduated, but also that the memory access time must be payed just once.

- **The vectorization percentage of the programs.** This measure will also affect their performance results. It is intuitive that the addition of a vector unit will provide some performance gain to those programs that can take advantage of it, as less instructions and operations must be carried out. Therefore, the greater the use of the vector unit, the higher the performance gain that can be obtained, to a certain extent. All in all, the greater performance gains will be obtained for programs with both, a high vectorization percentage and a large vector length with vector stride equal 1.

The performance study for the real memory system will include not only the overall performance data, but also a deep study about the behavior inside D-regions and S-regions.

We will compare the performance results against a superscalar architecture backed with a traditional cache based memory system. Of course, the SS architecture has also been properly scaled. This comparison allows us to make an evaluation of the proposed architecture in terms of performance feasibility.

In short, these scalability studies will show that:

- The SSV architecture scales very well as more memory and computing resources are added.
- The SSV architecture reaches higher performance values than the traditional SS architecture.
- The SSV architecture achieves higher parallelism inside D-regions than the SS architecture. Performance inside S-regions is better for the SS architecture. Its contribution to the overall performance is determined by the relative weight of D- and S-regions in the whole programs.
- For vectorizable programs, the overall performance is mainly determined by the behavior inside D-regions.
- Numerical programs put a higher pressure on the main memory, and the CB memory model reacts much better to this pressure with a lower main memory traffic and higher hit rates.

5.2 MACHINE CONFIGURATIONS

The scalability studies previously introduced will be carried out over a set of configurations. Table 5.1 shows the architectural parameters of all the configurations for both, the SSV and SS architectures. The upper part of table 5.1 presents the SSV configurations and the lower part shows the SS configurations. A machine configuration, either

Superscalar with Vector Unit (SSV)												
MEMxFLOPS	1x2	2x4	4x4	4x8	4x16	8x4	8x8	8x16	16x4	16x8	16x16	16x32
Fetch/issue/grad	4	-	4									
# Mem. ports	1x1w	-	1 x 4 words			1 x 8 words			1 x 16 words			
# VEC FP units	2x1	-	2x2	2x4	2x8	2x2	2x4	2x8	2x2	2x4	2x8	2x16
# VEC INT units	2x1	-	2x2	2x4	2x8	2x2	2x4	2x8	2x2	2x4	2x8	2x16
ROB size	64	-	64									
L/S queue	32	-	32									
BTB/Lbranch	0.5/4K	-	0.5/4K									

Superscalar (SS)												
MEMxFLOPS	1x2	2x4	4x4	4x8	4x16	8x4	8x8	8x16	16x4	16x8	16x16	16x32
Fetch/issue/grad	4	8	-	16	-	-	-	24	-	-	32	48
# Mem. ports	1x1w	2x1w	-	4x1w	-	-	-	8x1w	-	-	16x1w	16x1w
# FP units	2	4	-	8	-	-	-	16	-	-	16	32
# INT units	2	4	-	8	-	-	-	16	-	-	16	32
ROB size	32	64	-	128	-	-	-	256	-	-	256	512
L/S queue	16	32	-	64	-	-	-	128	-	-	128	128
BTB/Lbranch	0.5/4K	1/8K	-	2/16K	-	-	-	4/32K	-	-	4/32K	4/32K

Table 5.1 Configuration parameters for the Superscalar with Vector unit (SSV) and Superscalar (SS) architectures.

SSV or SS, is defined by two parameters **MEMxFLOPS**, where **MEM** is the total number of 64-bit words that can be moved between memory and the register file in each cycle, and **FLOPS** is the total number of floating point results that can be performed in every cycle. For example, the SSV-8x16 configuration defines an SSV architecture that is able to read 8 64-bit words from memory each cycle and can generate a total of 16 floating point results per cycle (8 results per cycle in each of the two functional units).

The SSV architecture has been built starting from a basic superscalar core that is able to fetch, decode, issue and graduate four instructions per cycle (configuration SSV-1x2 in table 5.1). This basic configuration has a unique memory port which provides one 64-bit word in each cycle, two floating point functional units, and two integer functional units, each one operating one result per cycle. The **NxM** name, for the FP and INT vector units, means that there are **N** different functional units, each one able to operate **M** data in each cycle. In our case **N** always equals 2, for the SSV architecture, as there are two different functional units of each type (INT and FP). In each case, we will be varying their widths, that is, the number of data that can be simultaneously operated.

The rest of parameters that define the superscalar core of the SSV architecture have been configured taking into account the nature of vector programs and also knowing that this superscalar core will remain the same as the SSV architecture is scaled. It is important to provide enough entries in the reorder buffer due to the large number of cycles that vector instructions may take to execute. As an example, imagine a SSV configuration with 2 functional units, each one able to operate one data in each cycle, and maximum vector length of 128 elements. In this architecture, a vector add instruction will take 128 cycles to execute, once the functional unit has been assigned to the instruction. If the following instructions depend on the vector add that is being carried out, and no independent instructions can be found to be issued to the other functional unit, the processor will be stopped for many cycles. Dependences between consecutive vector instructions have an important negative effect because they cause a sequential execution of the instructions, even making some of the functional units be idle during many cycles. Therefore, it is important to provide the superscalar core with some additional aggressiveness. This aggressiveness will allow the processor to look further down the instruction stream for instructions that can initiate their execution without needing the results produced by the other instructions which are being executed. For that reason the size of the Reorder Buffer has been set to 64 entries.

As the SSV architecture is scaled, and therefore is able to process more and more data in each cycle, the effect previously discussed diminishes, and the additional aggressiveness of the superscalar core does not determine so much the performance behavior of the vector code. Following the previous example, imagine that the configuration in this case has two functional units and each of them can now process sixteen data in each cycle. In that case, the vector add instruction with 128 vector elements previously mentioned would just take 8 cycles to be executed. Despite the fact that, intuitively, it might seem that the aggressiveness of the superscalar core can be reduced as the configurations are scaled, this is not so. The reason is that we must guarantee that the scalar code of the vector programs execute fluidly, without becoming a performance bottleneck. Therefore, the performance differences between two of these configurations will come from the difference in the width of the functional units and the memory port.

In order to select the configuration set that we study in this chapter, we define what the relationship between the **MEM** and **FLOPS** parameters of a certain configuration should be, so that we will obtain the best performance/cost rate. In this sense, it is important to note that the connection with the memory system determines the performance answer in many programs. As we will see in the next chapter, in some of the programs, the performance is directly related to the bandwidth of the memory system. Moreover, it is more expensive to increase the **MEM** parameter than the **FLOPS** one. Therefore, we have designed the configuration set by fixing the **MEM** parameter and exploring the **FLOPS** parameter. For example, we have set **MEM** to 4, and we have explored **FLOPS** equal 4, 8 and 16, thus setting configurations SSV-4x4, SSV-4x8 and SSV-4x16, respectively. The most aggressive configuration is SSV-16x32, which can read/write from memory 16 64-bit words in each cycle, and perform 32 floating point results every cycle (16 results in each floating point functional unit). This way of generating the configuration set allows us to study the effect in performance of improving the computing power, as well as the effect of improving the memory power.

In the SS architecture, scaling the different configurations requires not only scaling the **MEM** and **FLOPS** parameters but also properly scaling the superscalar engine. The simpler configuration is SS-1x2, as can be seen in table 5.1. In this configuration the processor is able to read one 64-bit word from memory in each cycle, and it can process two floating point results in each cycle. This basic configuration is able to fetch, decode, issue and commit four instructions in each cycle. Its reorder buffer has 32 entries and the load/store queue has 16 entries. The branch predictor is configured with a 0.5 K entries branch target buffer (BTB) and the size of the gshare prediction table (Lbranch) is 4K. This configuration approximates a MIPS R10000 processor and is the basic configuration that we have used in this study.

As the SS configurations are scaled, we add more independent memory ports and functional units. For example, the SS-4x8 configuration, shown in table 5.1, has 4 independent memory ports, each one able to move one 64-bit word in each cycle, and 8 floating point and 8 integer functional units, all of them independent. Therefore, 20 results can be generated in each cycle. Naturally, fetch, decode and issue must be scaled to be able to feed these execution resources. Therefore, this configuration can

fetch, decode, issue and graduate 16 instructions in each cycle, its reorder buffer has 128 entries, the load/store queue has 64 entries and the branch predictor has a BTB of 2 K entries and a Lbranch size of 16K.

It is important to note the sharp difference between SS and SSV in each pair of configurations that has the same total memory and computing power, that is, any configuration pair that has the same **MEMxFLOPS** name. For example, for the pair SS-4x8 and SSV-4x8, the SS-4x8 configuration has 4 independent memory ports. Each port is able to move one 64-bit word in each cycle, from different memory instructions, so that they can potentially access different memory locations. In contrast, the SSV-4x8 configuration moves four 64-bit words that are sequentially located in memory. These four words correspond to a unique vector memory access. The SSV only needs a single port, of the appropriate width, while the SS architecture needs multiple ports, with the consequent increase in cost and complexity.

The SS-4x8 configuration also has 8 integer functional units and 8 floating point functional units, all of them independent of each other. Once again, these floating point units operate according to 8 different instructions that must be allocated in the reorder buffer, must be assigned to one of these functional units and must be issued for their execution. Although this scheme is more flexible, it is more expensive and complex to build. Its SSV counterpart can also yield 8 floating point results per cycle (4 results in each floating point functional unit). However, these results come from the execution of two vector instructions (one of them in each functional unit) that take their input data from a vector register. In the SS configuration, all functional units are independent, while in the SSV configurations they are not. As shown in figure 4.1, page 109, this is an important difference. We could see in that figure that the SSV architecture has only two floating point functional units (F0 and F1) that are replicated **K** times. The important point is that, from the point of view of the instruction issue logic, there are only two functional units to which scalar and vector instructions can be issued. The **K** replicas of the functional units are simultaneously operated, using the same control lines. All of them execute the same operation, in the same cycle, but using different input data. Each one of the **K** replicas takes its input data from one of the **K** lanes of the

Vector Register File (VRF). Therefore, the number of lanes of the VRF is determined by the configuration that we are considering in every case.

Regarding the superscalar core, and following the same example, we realize that while in the SSV architecture the superscalar core is quite modest and remains unchanged along all the configurations, in the SS architecture it is necessary to scale it in order to take advantage of the extra functional units and memory ports that are added to the processor. This fact means that the SS architecture devotes an ever increasing chip area to the processor control. Increasing the computing and memory power by adding more and more functional units leads us to an mandatory increase of the processor fetch, decode, issue and commit facilities in order to be able to exploit these extra functional units and memory ports. As the amount of area that the dispatch logic needs increases quadratically with the dispatch capacity [PJS97], it is important to pay attention to both features, the performance that is obtained in return, and the performance/cost rate of the different configurations.

The conclusion is that, even though two equally named configurations have the same total memory and computing power, they differ significantly in complexity terms. The SS architecture is more complex than the SSV architecture, for the same configuration. The reason is not only the different superscalar core of both architectures, but also the fact that the independent functional units and memory ports of the SS architecture are more complicated to build. This fact must be taken into account whenever a performance comparison between the SS and SSV architectures is carried out, thus avoiding to think that two SS and SSV configurations with the same name are exactly equivalent.

We finish this section recalling that we will evaluate in this chapter the performance behavior by using the hybrid set of benchmarks already discussed in chapter 2. We will use the EIPC performance metric also discussed in that chapter.

5.3 PERFECT CACHE AND SCALABILITY

We start by exploring the performance and the scalability of our proposed architecture assuming an ideal memory system. As stated before, this study will allow us to explore the maximum performance results that could be obtained with a SSV architecture as more resources are added to the architecture. We simulate a perfect memory by assuming that the cache is perfect, that is, all memory accesses hit in the L1 and multiple accesses are served without conflicts. All scalar memory accesses take one cycle. Vector memory accesses are served in $\lceil VL/B \rceil$ cycles, where B is the bandwidth of the single memory port, that is, 1, 4, 8 or 16 words per cycle.

5.3.1 General Performance

Figure 5.1 shows the performance results for the SSV architecture when using vector lengths of 128 and 16 elements. The performance results for intermediate values of maximum vector length will be comprised between these two. It also shows the performance results for the SS architecture. It is important to note that the scale in the y-axis is different for the different programs, a fact that must always be taken into account for comparison purposes.

Analyzing the SSV performance results, we realize that all programs improve their performance as more resources are added to the processor. This is especially true for the 128-length vector registers configuration, but, nevertheless, the 16-length vector register configurations still achieves convincing performance gains with very modest investments in chip area (only 4KB for the Vector Register File).

We also observe that numerical programs scale much better than multimedia programs. In general, the four multimedia benchmarks (*Gsm Encode*, *Jpeg Encode*, *Jpeg Decode* and *Epic*) present modest performance results when compared to the numerical codes. This seems to contradict the common belief that multimedia workloads present huge levels of data level parallelism [CEV99]. Our analysis shows that, while this is true in raw kernels (such as the basic `idct` algorithm), several problems arise when we look at complete applications. The main reason is the vectorization percentage, dis-

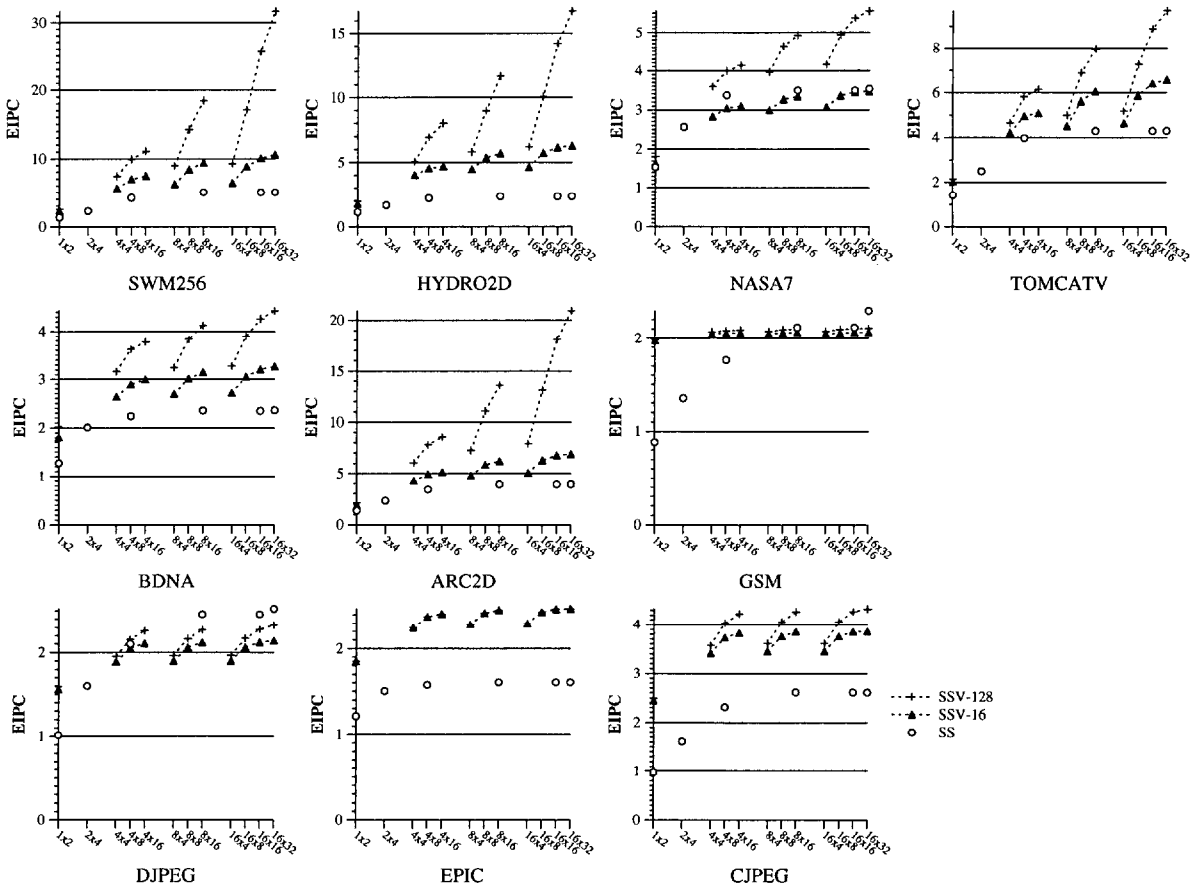


Figure 5.1 Ideal Performance of the SSV architecture, for vector lengths 128 and 16, and comparison with the ideal performance of the SS architecture.

cussed in section 3.7, page 67. We could see there that the vectorization percentage is smaller in the multimedia programs (less than 70%, hardly achieving 40% in two cases). Meanwhile, numerical programs reach quite higher vectorization percentages (over 75%, being higher than 95% in three cases). As the multimedia programs are less vectorizable, when the SSV architecture is scaled there is a smaller instruction percentage that can benefit from these additional resources, and the overall performance increase is small. Moreover, as the superscalar core of the SSV architecture remains constant across all the configurations, the execution of the non-vectorizable pieces of the programs does not contribute to improve performance while configurations are scaled. In short, multimedia programs are characterized by relatively low vectorization percentages, and therefore, as we scale the system, the non-vectorizable fraction of code

becomes the real performance bottleneck due to the lack of enough issue rate and scalar resources [Amd67] [PH96].

In figure 5.1, we can also see that there is a performance loss when going from 128-length to 16-length vector register configurations. This performance loss is caused by the increase in the total number of instructions and operations executed when decreasing the maximum vector length (as discussed in section 3.8, page 77). As more instructions are executed, a higher number of cycles is needed to execute the whole program. It is important to note that we are using the EIPC metric, which makes the performance loss even more obvious, as discussed in section 2.8, page 51.

The performance loss from 128-length to 16-length configuration is substantial in numerical programs. The reason is that, as discussed in section 3.8, page 77, decreasing the vector length increases the total number of instructions and operations executed, especially in numerical programs. Moreover, the performance loss enlarges as the number of lanes of the functional units is increased (keeping constant the width of the memory port). This fact reflects that the execution time of the vector instructions decreases while the execution time of the scalar part does not change. Therefore, by Ahmdal's Law [Amd67] [PH96], the execution of the scalar part limits the overall performance.

The difference in multimedia programs between both vector lengths is by contrast quite small. The extreme case is program *Epic*. As discussed in section 3.7, page 67, this program uses vector length of 16 elements in almost all its vector instructions. Therefore, using a maximum vector length of 128 elements does not influence the performance results at all, because the 112 extra elements of the vector registers are not used. Thus, the performance graphs for the 128-length and 16-length configurations are almost identical. This behavior will be seen again in the following chapters.

As far as the *Gsm Encode* program is concerned, the majority of its instructions use vector length 128 (as shown in figure 3.2 in page 70), so that using 16-length vector registers increases the total number of instructions, operations, and cycles of program execution. However, the low vectorization percentage of this program implies that this increase will not have important consequences in its overall performance.

A possible solution to the modest results in multimedia benchmarks would be the adoption of new instructions based on sub-word level oriented parallelism, such as MMX [PW96] [CEV01], but translated to true vector operations. This would allow increasing the effective vector length and reducing the impact of undesired recurrences in reduction operations. The undesired side effect would be the requirement of high efforts on compilation techniques development.

Let us now observe that keeping constant the memory port width, and increasing the number of lanes of the functional units, an important increase in the performance results is obtained. This is true for the different memory port widths that have been studied, that is, 4, 8 and 16 words. In the latter case, for 16-word memory port width, the graphs start to flatten. We can also observe the performance effect of keeping constant the number of lanes in the functional units and increasing the memory port width by drawing an imaginary line between the performance points for 4x4, 8x4 and 16x4 configurations, for example. We observe that the increase is very slight in comparison to the increase when more lanes are added to the functional units. The reason is that, as the memory system is perfect, it will bring 4, 8 or 16 words in very few cycles. As the number of lanes is not increased, we can not process the increasing number of elements that the main memory delivers, so we can not benefit from receiving more and more elements per cycle.

We now turn to the comparison with the SS architecture. In figure 5.1, we can clearly see that our proposed architecture achieves similar performance to the basic superscalar version for the 1x2 configuration in almost all programs. As we increase the number of processor functional units (or the number of lanes in the case of the vector functional units), the SS architecture performance levels off. Meanwhile, our proposed processor improves at a much higher rate, with programs *Nasa7*, *Jpeg Decode* and *Gsm Encode* being the exception. Although the SS architecture has more fetch, decode, issue and graduate facilities, it is not able to decrease the number of execution cycles that will probably be determined by the dependences between the program instructions due to the way in which programs are expressed by the compiler. By contrast, in the SSV architecture, the execution of a vector instruction takes many cycles. Therefore, the functional unit is busy during all those cycles and the processor has enough time to

find out an independent instruction to be issued to the functional unit, as soon as it gets free. In the SS architecture, however, an instruction takes just one or two cycles to be executed, quite few cycles to ensure that an independent instruction, ready to be issued for execution, will be found. Thus, it is likely that the functional unit can stay idle during a certain number of cycles.

As mentioned before, three programs do not follow the same behavior. Looking at each program individually, we can point out the following reasons to their different behavior. In the *Gsm Encode* program, the SS performance results do not overcome the SSV graphs until the 8x16 configuration. At this point, the *Gsm Encode* program, which has more than 60% of scalar instructions, takes more advantage of the improvements in the superscalar core than in the increase in the number of lanes of the SSV functional units. Note that in the 4x8 configuration still happened the opposite situation. This is another proof of the importance of the scalar code in low vectorizable programs. A similar behavior happens in *Jpeg Decode* program. *Nasa7*, however, presents a slightly different behavior. The 128-length SSV architecture attains a higher performance than the SS architecture. However, when the maximum vector length is reduced down to 16 elements, the total number of execution cycles increases and becomes larger than the number of execution cycles in the SS architecture. The difference in the total number of cycles between the 16-length SSV architecture and the SS architecture is, in fact, quite small, thus providing similar EIPC results.

As a conclusion, the scalability of the SSV architecture is much better than the plain superscalar machine, and, furthermore, this scalability is achieved using a much lower control complexity. With a 4-way out-of-order engine, we are able to successfully feed up to 32 floating point units and sustain a large fraction of peak performance.

5.3.2 Performance Breakdown by Regions

While the previous section carried out a study about the overall performance and scalability of the SSV and SS architectures, this section presents an in deep specification of the same study, discussing the performance behavior inside D-regions and S-regions.

Program	% Ops executed inside D-regions	% Ops executed inside S-regions	Vectorization %	Average size of D-regions	Average size of S-regions
Swim256	99.99	0.01	98.77	380988	21
Hydro2d	99.11	0.89	95.64	44420	396
Nasa7	99.05	0.95	75.62	217266	2077
Tomcatv	83.78	16.22	82.85	1178606	227814
Bdna	88.88	11.12	66.78	82990	10381
Arc2d	99.57	0.43	97.53	135348	585
Jpeg Decode	60.43	36.57	61.00	18146	11877
Epic	62.12	37.88	44.01	5824	3550
Jpeg Encode	78.77	21.23	70.05	1941	523
Gsm Encode	38.90	61.10	36.95	1098	1725

Table 5.2 Percentage of operations executed inside D- and S-regions, vectorization percentage and Average size of D- and S-regions (in operations), extracted from the vector programs. Average size is defined as the total number of operations executed inside S-/D-regions divided by the number of S-/D-regions.

Recall from chapters 2 and 3 that every program has two types of regions, called D-regions and S-regions. D-regions contain those pieces of code that can be vectorized, and S-regions contain pieces of code that are not amenable to be expressed using vector instructions. We measure performance inside S-regions and D-regions by using the EIPC S-reg and EIPC D-reg metrics, defined in section 2.8, page 51. From those definitions, it is clear that measuring performance inside D-regions and S-regions separately will allow us to study the performance behavior in the pure vector and pure scalar program zones. Thus, we can better understand the amount of parallelism that is extracted in each type of region, and it will help us to detect the performance bottlenecks of the different programs.

The relative contribution of D-regions and S-regions to overall performance depends on the average size of D-regions and S-regions, in terms of operations executed. That information was discussed in section 3.9.1, based on data shown in table 3.13. In order to ease readability, the table is reproduced in table 5.2 in this chapter.

Figure 5.2 shows the performance results inside D-regions, that is, the SSV EIPC D-reg measure previously defined. From this figure, we can state that all programs achieve higher performance results as the SSV architecture is scaled when only the data-parallel regions are considered. It can be detected because the values in the y-axes

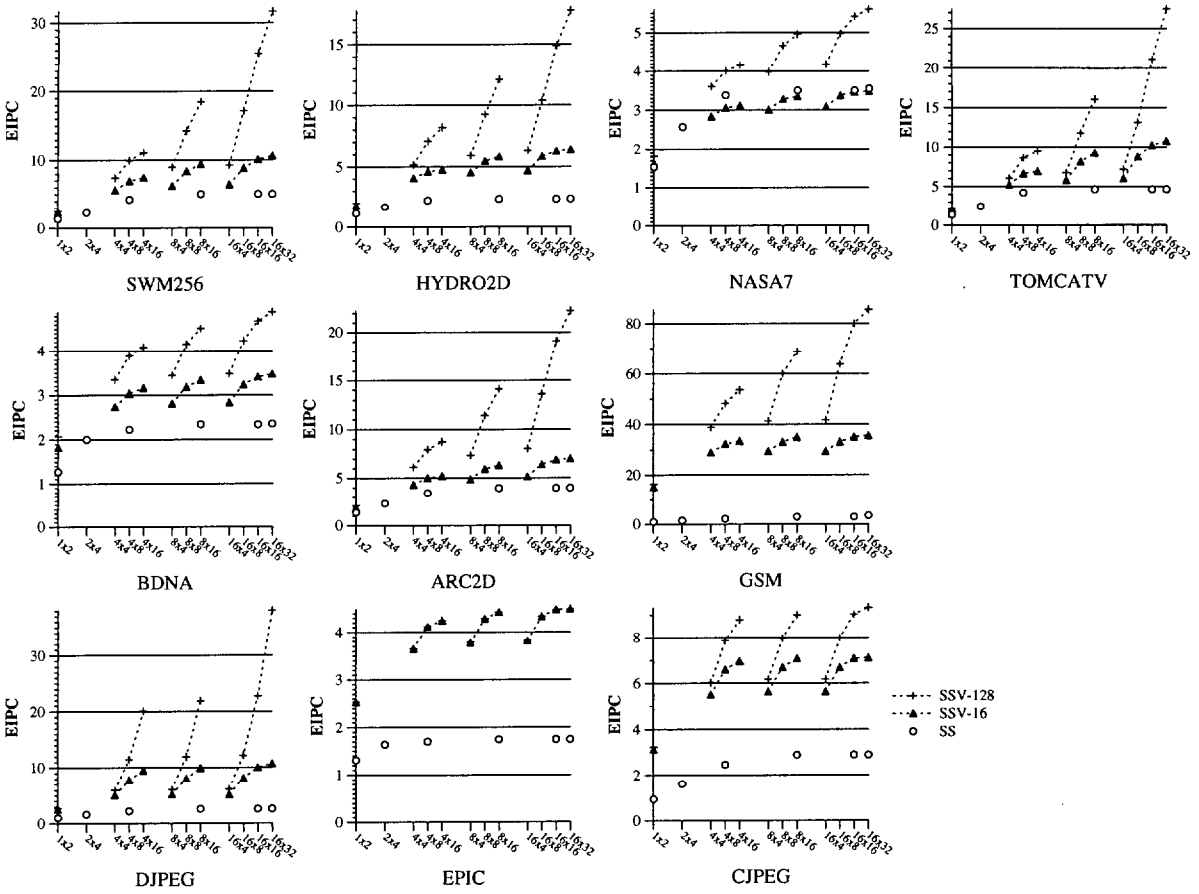


Figure 5.2 Ideal Performance inside D-regions for the SSV architecture, for vector lengths 128 and 16, and comparison with the ideal performance inside D-regions of the SS architecture.

for the different graphs have moved to higher values in most cases. Some programs, like *Swim256*, hardly show any difference when compared to the global ideal performance because they are 99% vectorizable, so that the effect of the S-regions in the overall performance is negligible. Nevertheless, programs like *Tomcatv*, *Gsm Encode*, *Jpeg Decode*, *Epic* and *Jpeg Encode* show quite important performance improvements, being spectacular in some cases. For example, that is the case of programs *Tomcatv*, which in the overall performance obtained a maximum EIPC value of 9.7 and now reaches as much as 27.5, *Gsm Encode*, moving from 2.1 to 85.6, and *Jpeg Decode*, which goes from 2.3 to 38.1. Most of these programs have low vectorization percentages. So, it is normal that, adding up the effects of S-regions and D-regions, the generated overall performance is a rather smaller value than that obtained inside D-regions.

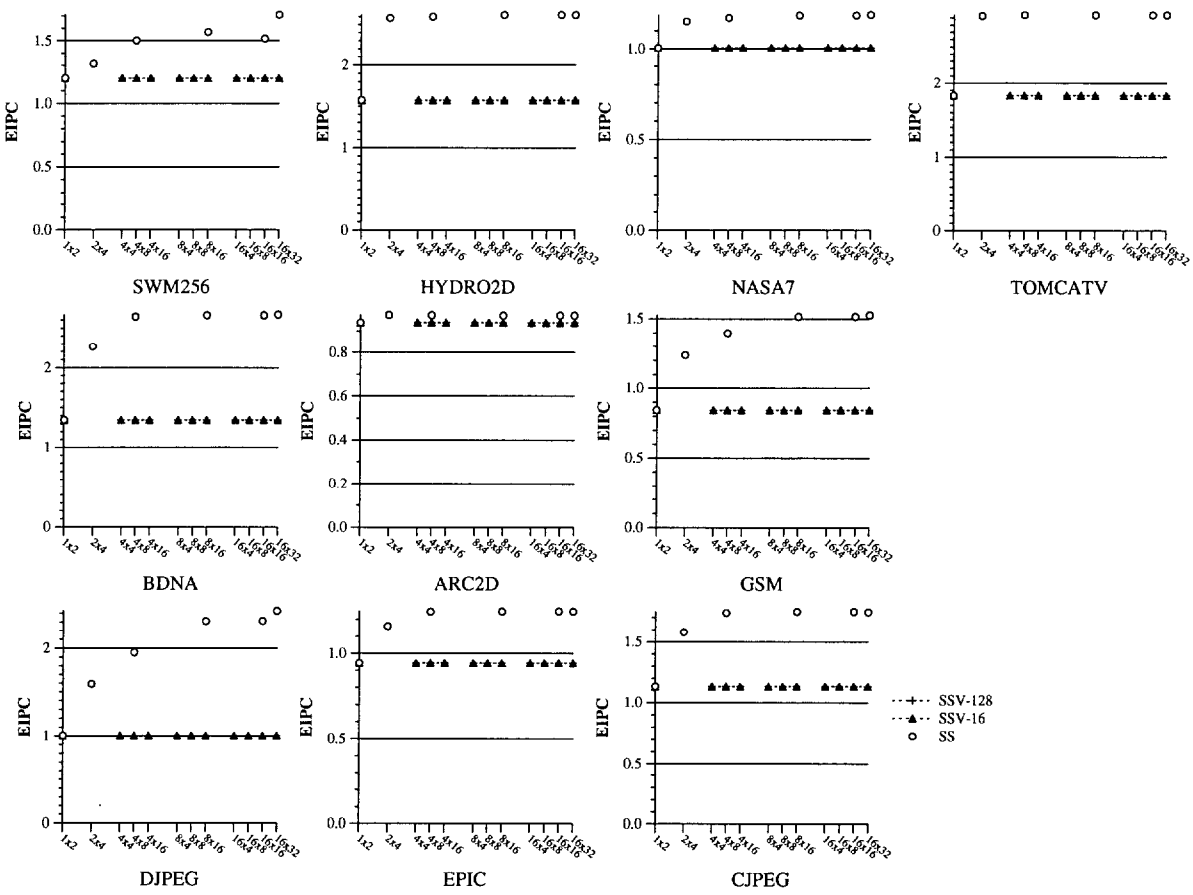


Figure 5.3 Ideal Performance inside S-regions for the SSV architecture, for vector lengths 128 and 16, and comparison with the ideal performance inside S-regions of the SS architecture.

The rest of the programs barely modify their behavior with respect to that shown in the overall performance, shown in figure 5.1. *Nasa7* also behaves in the same way, and it becomes the only program in which the SS architecture equals the performance results of the SSV architecture. The performance values obtained inside D-regions are very similar to those shown in the overall performance (figure 5.1). We can observe in table 5.2 that this program executes 99% of their operations inside D-regions, but it is only 75% vectorizable. Therefore, 25% of operations inside D-regions are scalar, which means that D-regions are polluted with a large number of scalar instructions that are executed in a 4-way superscalar core in the SSV architecture, thus reducing the attainable EIPC.

On the other hand, figure 5.3 shows the ideal performance results inside S-regions. These values have been obtained from a superscalar simulation of the different SSV configurations, extracting performance results inside S-regions. It means that the processor configuration has been scaled in the SSV way, so that regardless of the scaling the scalar code is always executed in a 4-way superscalar core with the same configuration parameters. The analysis of the EIPC achieved inside S-regions, shown in figure 5.3, leads us to the following considerations. Starting in the 1x2 configuration, the SSV and SS performance results are very similar. Nevertheless, from that configuration upwards, the SS architecture results are clearly better than the SSV ones. The performance increases of the SS architecture, however, are rather small, and the performance values achieved hardly range from 0.8 to 2.9 in the best case.

This kind of behavior was expected. While in the SS architecture the superscalar engine is scaled until fetching 16 instructions in each cycle, in the SSV architecture the superscalar core remains unchanged along all the configurations. Moreover, the SSV results do not change as the architecture evolves because the increases in the memory port width or the number of lanes of the functional units are not used during the execution of the scalar code. Similarly, the SSV results do not change when moving from vector length 128 to 16. Therefore, one could argue that the important characteristic is the configuration of the superscalar engine, as not all programs can be vectorized, and consequently, the vector functional unit would not be used in all cases. Nevertheless, as shown in figure 5.3, the EIPC curves of the SS architecture tend to flatten in quite modest configurations, after a slight initial increase. The conclusion is that, although a certain out-of-order execution is required, the more aggressive superscalar core does not necessarily provide the better performance. Moreover, as more and more transistors fit in the same chip area, it is necessary to decide the way in which this extra capacity will be used. So, once the out-of-order engine has reached the bound from which the EIPC is not increased, the addition of a vector unit will help the exploitation of additional parallelism.

From figures 5.2 and 5.3, we can also conclude that although vector functional units can achieve high performance measures inside D-regions, if the relative weight of D-regions in the total program is small, the global EIPC will be dominated by the performance

values obtained inside S-regions, which do not reach 3, even for the very aggressive 16x32 configuration.

5.3.3 Data Parallelism Inside Vector Regions

The performance measure calculated up to now is Equivalent IPC (EIPC), as defined in section 2.8, page 51. This measure was defined in order to be able to compare the performance between superscalar and superscalar+vector architectures in a trustworthy manner. As stated before, vector programs execute less instructions and operations than scalar programs and we had to take into account this fact when performance is measured and compared.

Although the EIPC measure overcomes that problem, it is not an appropriate metric if we want to measure the actual data parallelism that each architecture exploits inside D-regions. To do that, we compute the "operations per cycle" rate (OPC) within each D-region. This measure is calculated for pure vector and scalar programs, that is, no hybrid programs are used. With this measure, we do not take into account the different number of instructions and operations that each architecture provides. We just look at the number of operations that each architecture is able to deliver per cycle, inside D-regions.

Figure 5.4 compares the distribution of OPC for the SSV and the SS architectures. We present the lower and upper bounds of the configuration set, that is, 1x2 and 16x32 configurations. For the SSV architecture we show the results for the 128-element and 16-element vector registers.

Each bar in figure 5.4 shows the percentage of the total number of operations that are executed at a certain OPC. For example, this figure shows that, for the SS-16x32 configuration, about 70% of D-region operations of *Swim256* are executed at a rate of 5 or more operations per cycle. The greater the OPC, the darker the corresponding bar color. We can compare two bars at a glance and determine which one reaches a greater OPC just looking at their colors. For example, if we compare in program *Swim256* the bars corresponding to the SSV-1x2 architecture, with 128-element and 16-element

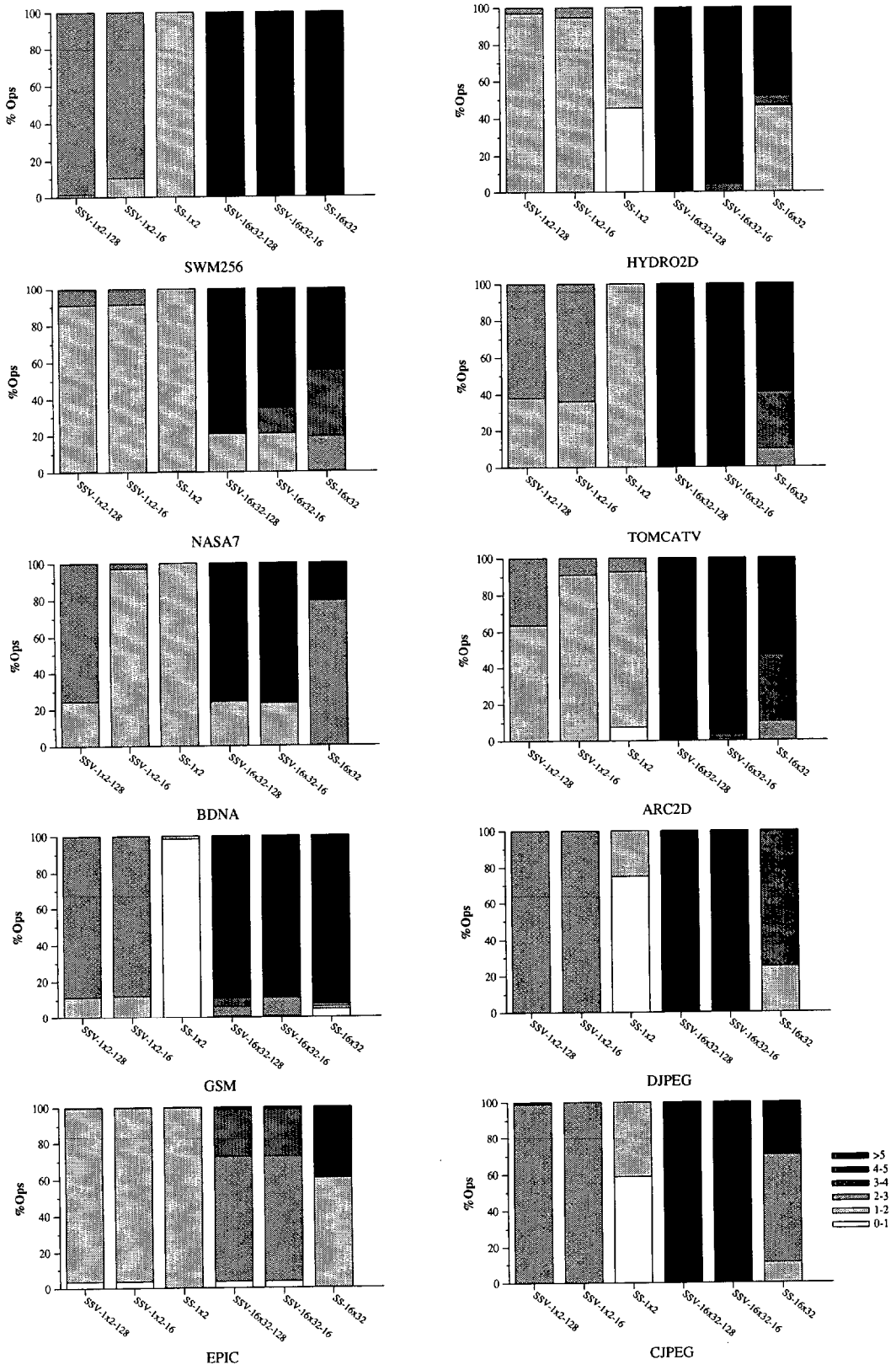


Figure 5.4 Distribution of operations percentage executed within a certain OPC range, inside D-regions.

vector registers, we can see that the main difference is the lighter lower region in the 16-element vector registers bar. It means that, with 128-element vector registers, the SSV-1x2 architecture is able to execute almost all operations in the [2-3] OPC range, while, with 16-element vector registers, the processor executes 80% operations in that OPC range, and the rest in the [1-2] OPC range.

Several behaviors can be observed from figure 5.4:

- As we have already discussed in the previous section, as the processor configurations are scaled, both, the SSV and the SS architectures, are able to exploit more and more data level parallelism, which can be seen because the bar color becomes darker when going from the 1x2 to the 16x32 configurations.
- Moreover, the SSV architecture exploits much more data level parallelism than the SS architecture, given the darker bars for the SSV architecture. The differences are especially large for *Gsm Encode* as it is not able to reach the same performance levels than the SSV architecture neither in the 1x2 nor in the 16x32 configurations. The same happens to *Jpeg Decode*, *Hydro2d*, and the rest of programs, although in a lower extent. Meanwhile, the SSV architecture reaches large values of OPC, yet it only needs to issue up to three vector instructions in parallel (two floating point functional units and one memory port).
- Many programs, like *Tomcatv*, *Gsm Encode*, *Jpeg Decode*, *Epic* and *Jpeg Encode*, hardly differ in the OPC that are able to exploit inside D-regions when the vector length is changed from 128 to 16. The rest of the programs undergo some changes in the exploited OPC when the vector length is decreased. Comparing these results with those shown in figure 5.2, we can see that, in figure 5.2, decreasing the vector length also decreased performance. The different behaviors come from the different measures used in these figures (OPC and EIPC). While OPC does not change, EIPC decreases as it only takes into account the increase in the number of execution cycles, but not the increased number of instructions and operations executed when the vector length is decreased.

As a general conclusion of the scalability study under perfect memory, we can remark that the SSV architecture scales very well as more memory and computing resources are added to the processor. It also reaches higher values of parallelism than the SS architecture with a lower cost and control complexity. The analysis inside regions shows that the SSV architecture achieves large values of parallelism inside D-regions, although their effect in the overall performance is determined by the relative importance of D- and S-regions in the whole program. Inside S-regions, the SS architecture reaches higher results, although it does not scale as well as it would be expected, taking into account the resources invested in each configuration.

5.4 REAL MEMORY HIERARCHY

Once we have analyzed the potential performance and scalability of the SSV architecture, we turn now to the question of the performance under a real memory hierarchy. We will study the two cache architectures described in section 4.3, page 116, CA and CB. We will look at their performance studying, first, the cache hierarchy efficiency for both models (in terms of total memory traffic and hit ratio). Then, we will make a study about the reasons that make the vector cache stall, and, finally, we will present some performance results. These results will also be compared to the behavior of a SS architecture backed with a typical cache hierarchy.

5.4.1 Processor Configurations

The processor configurations that we will study are a subset of those previously defined in table 5.1. There are two reasons for using a restricted number of processor configurations from that used in the scalability study in section 5.3. On one hand, we consider that it is time to shorten the processor configurations spectrum to a set of feasible configurations, in terms of mem/flops ratio. In fact, the overall objective of this work is to study the SSV architecture and to give a proposal of a SSV architecture with its memory hierarchy. Shortening the configuration set will finally lead us to the final proposal of a feasible SSV architecture. The question is how to make the selection of a subset of configurations from those already studied in section 5.3. The

results shown in that section achieved the better performance whenever the number of floating point operations that can be carried out by the architecture doubles the number of words that can be simultaneously moved to/from the memory, that is, for the SSV-1x2, SSV-2x4, SSV-4x8, SSV-8x16 and SSV-16x32 configurations. Although some of these configurations may be inviable to be built, especially for the SS architecture, our purpose is to show the performance behavior as both architectures are scaled.

On the other hand, practical feasibility reasons have also led us to shorten the configuration set. While results in section 5.3 have needed twenty eight simulations for each program (yielding a total of 280 simulations), the study that we are carrying out in this section would need sixty simulations for each program (thus, requiring 600 simulations in total). Considering the amount of time that is necessary in order to finish a set of simulations, we think that we must reduce the configuration set. Of course, this reduction has been done according to the criterion already commented, which will lead us to performance results where the general trends can be identified, without losing significant information.

5.4.2 Memory Hierarchy Configurations

As stated before, we will study the processor behavior when a real memory hierarchy is introduced. In chapter 4, we have defined the design of the memory hierarchies that will be studied. There are two different models, CA and CB, both of them based on the introduction of a vector cache. Figure 5.5 presents the CA and CB models again, including some additional information about the specific parameters that will be used in this study. The difference between both models comes from the location of the vector cache. While, in the CA model, the vector cache is in the first cache level, in the CB model, the vector cache is located in the second cache level. An additional difference between both models is the direct connection between the processor and the vector cache in the CB model. While in the CA model all data that the processor accesses are in the first cache level, in the CB model, scalar data are accessed in the first cache level while vector accesses are made to the second cache level [Hsu94] [Sha99]. The underlying idea in the CB model is the decoupling between where scalar and vector data are located, and accessed, in the cache hierarchy.

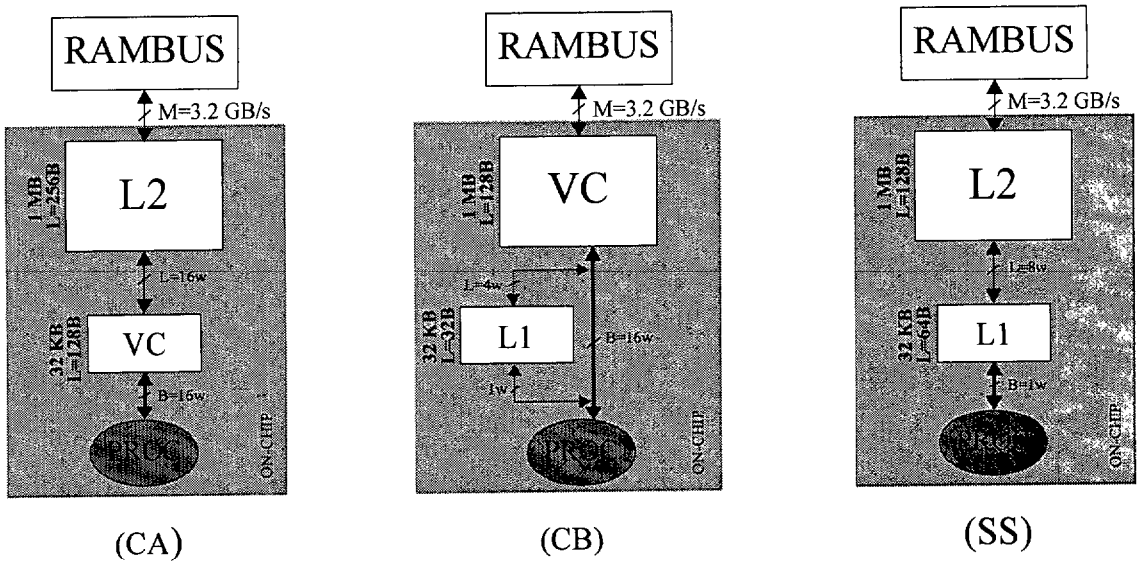


Figure 5.5 Memory hierarchies studied for the CA and CB models of the SSV architecture and for the SS architecture.

Figure 5.5 also shows the cache hierarchy that will be used in the SS architecture and some of the configuration parameters. We can see in this figure that the SS architecture has been configured following a Mips R10000 processor. There is only a memory port connecting the processor with the cache hierarchy. In fact, this is a common characteristic to the three cache models (CA, CB and SS). Increasing the L1 cache bandwidth does not have much sense unless we increase the number of processor memory ports. We will analyze those enhancements in later sections.

The general parameters of the two memory hierarchies under study are summarized in table 5.3. This table shows three different memory configurations which correspond to the CA, CB and SS memory hierarchies, respectively. We will use a single memory configuration for the different processor configurations shown in previous sections. For example, we will use the same CA configuration for the different SSV configurations. We know that a special tuning of the memory configuration for each processor configuration would provide a more accurate result. We will make that tuning in the following chapter where we will study the memory hierarchy more deeply. However, at this point, we are

	CA		CB		SS	
	L1	L2	L1	L2	L1	L2
Size	32KB	1MB	32KB	1MB	32KB	1 MB
# Sets	256	2048	1024	4096	512	4096
Line Size	128B	256B	32B	128B	64B	128B
Associativity	1	2	1	2	1	2
Write Policy	WT	WB	WT	WB	WT	WB
Allocate Policy	NWA	WA	NWA	NWA	NWA	NWA
Latency (cycles)	1	4	1	4	1	4
Ld/St cycles	1/2	1/1	1/1	1/2	1/1	1/1
MSHR entries	8	8	8	8	8	8
WB depth/retire	8/4	8/4	8/4	8/4	8/4	8/4

Table 5.3 Cache Hierarchy Parameters: WT=write-through, WB=write-back, WA=write-allocate, NWA=no-write-allocate. *Ld/St cycles*, cycles required for a Load/Store operation. *WB depth/retire* number of Write Buffer entries/retire-at-X policy.

more concerned about general trends that can point out performance bottlenecks. The bottlenecks encountered will be addressed in the following sections.

The memory hierarchy parameters shown in table 5.3 were set by performing several studies, from which we would like to mention a few facts: (a) making the L2 4-way did not make much difference in any of the results presented; (b) the no-write-allocate policy for the L2 is substantially better than the write-allocate policy in at least half the benchmarks; (c) in the CB model, the size of the L1 (either 32Kb or 64Kb) has minimal impact on performance; (d) from the different variations that were tested, we selected the configuration that, on average, is best suited for the representative workload that we are studying.

Regarding the different parameters of the three memory hierarchies, it is worthwhile mentioning the following details:

- **Same cache sizes in all memory hierarchy models.** We have used the same cache sizes in order to make a fair comparison between different architectures. From the processor point of view we attach a 32KB first level cache, a 1MB second level cache and a main memory that is able to deliver 3.2 GB/s. The differences come from how these total sizes are then configured, that is, the line size, the number of

lines, the number of sets, etc. Although the internal configuration of a cache also influences the total size of the cache memory, we have not analyzed such details.

- **Different line sizes for the different memory hierarchies.** In the CA model, the vector cache is located in the first cache level. Both, scalar and vector memory instructions, access this cache, so that it is necessary to make a great amount of sequential data available in order to exploit the spatial locality of stride-1 vector memory accesses. For that reason, we have used 16-word sized lines. A 16-element vector memory access needs just one or two cache lines in order to be executed, while a 128-element vector memory access needs 8 or 9 cache lines. There is no much sense in providing 32-element cache lines because it would never be used when the vector registers are 16-element long. Moreover, as the total cache size is constant, providing larger cache lines decreases the number of different lines, thus worsening the exploitation of the temporal locality. In the L2 data cache, the line size must be at least twice the size of the L1 cache line. In the CB model, however, as only the scalar data are accessed in the L1 data cache, not so large cache lines are needed. So, we have set 4-word cache lines. In the L2 vector cache, as vector data are accessed, we need again large cache lines in order to exploit spatial locality. For the same reason as in CA model, we have set 16-word cache lines. Finally, in the SS architecture, after testing some configurations, the best configuration defines 8-word L1 cache lines and 16-word L2 cache lines.

- **Same associativity in all memory hierarchy models.** In the three memory hierarchies, we have used a direct mapped first level cache and a two-way associative second level cache. As stated before, improving the second level cache associativity did not introduce a great change in performance results, so that we preferred the cheapest option.

- **Latencies.** We have set the L1 cache latencies to 1 cycle and the L2 cache latencies to 4 cycles. In this way, we are assuming that both caches are on chip. Although this is not a realistic assumption for the more aggressive configurations, and especially for the SS architecture where the processor control is very complex and it may prevent from including the secondary cache on chip, we have done it for simplicity. In the following chapter, where we make a tuning of the memory

system configuration according to each processor configuration, we will study the effect of including larger secondary caches with higher latencies.

- **Load/Store cycles according to the cache type.** As stated in chapter 4, where the vector cache was introduced and studied, the vector cache takes twice as many cycles for a store operation than for a load operation. Therefore, these are extra cycles that the SSV architecture has to pay in each store operation, when compared to the SS architecture.
- **Same MSHR and WB structures in all memory hierarchy models.** Again, as a first approximation and in order to make a fair comparison, we have used the same MSHR and WB structures for the different memory models. However, vector memory accesses consume a large number of entries in these data structures (as many as the the number of cache lines involved in the access), so that they may fill up and stall the processor for many cycles. We would like to remark now this potential source of performance loss, because, in the following chapters, we will try to enhance performance by tuning the memory hierarchy configuration.

5.4.3 Cache Hierarchy Efficiency

As a first approximation, we will study in this section the behavior of the memory hierarchy from both, the traffic and the hit/miss point of view. In the following section we will study the reasons that make the vector cache stall. We will also look at performance in terms of EIPC, and we will introduce the comparison with the SS architecture.

It is important to remark that this study of cache hierarchy efficiency uses pure vector programs, rather than hybrid programs. The reason is that there is not much sense in measuring cache parameters inside S-regions and D-regions and then combining these two measures to give just a single number. When a pure vector program executes a S-region, the load and store instructions executed in that region change the state of the caches, so that, when the following D-region starts its execution, a different state is found. Some useful cache lines could be evicted due to conflicts, some other cache lines could also be loaded and these changes influence the execution of the following

D-region. Moreover, if we consider that, in the vector programs, the S-regions code has a low quality (as discussed in section 2.7, page 49, and stated in section 3.10, page 100), the total number of memory accesses will be higher, thus increasing the effect of polluting cache memories.

Similarly, in a pure scalar program, when a D-region is executed, the different way of accessing memory changes the cache state and influences the execution of the following D-region. Moreover, when the measures of S-regions and D-regions are combined, we realize that, for any pure scalar program, the state of the cache memories, at a certain execution point, has nothing to do with what would be their state at the same program point in the pure vector program. Therefore, this combined measure has no sense. Instead of that, we prefer to measure the behavior of pure vector programs. Although S-regions code has a low quality, global results are more coherent and will give us a lower bound of the measures.

The following subsections introduce the study of the main memory traffic and the hit percentage for the CA and CB cache hierarchies.

Memory Traffic Filtered

The first question we are concerned with is whether our workloads will take advantage of the caches or not. Traditionally, it has been claimed that vector workloads have low spatial and temporal locality, that this locality is well exploited by the large vector registers and that using a cache only gets in the way of memory accesses that end up accessing main memory regardless [KSF+94].

To answer this question, we have measured the percentage of 64-bit words that the caches are not able to filter and eventually reach main memory, that is, the fraction of 64-bit words that are moved to or from the RDRAM array over the total number of 64-bit words requested by the processor core. Figure 5.6 presents the results for the CA and CB cache models for both, 128-element and 16-element vector registers. In that figure, we can see, for example, that for *Hydro2d* program and in the CA model,

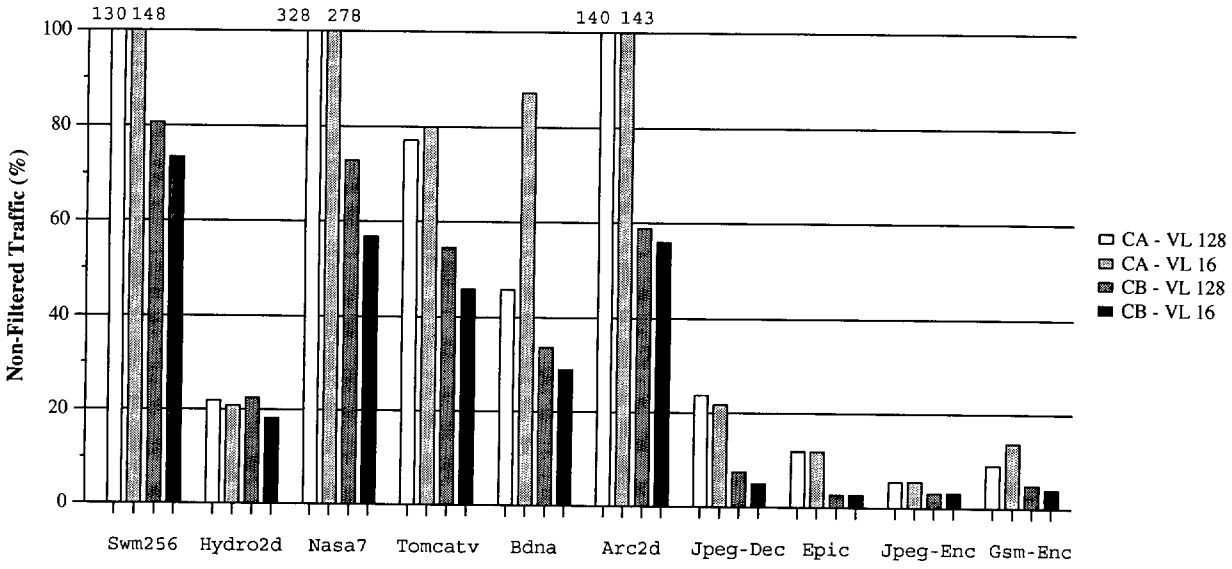


Figure 5.6 Non filtered traffic.

only about 20% of all the 64-bit words requested by the program end up being served by the RDRAM array. The other 80% is served by either the L1 or L2 caches.

In figure 5.6 we can see a special situation for *Swim256*, *Nasa7* and *Arc2d*, since the ratio is larger than 100%. This means that the amount of traffic with the main memory is larger than the number of words that the processor actually requests. The reason is that there is a lot of pollution in the cache lines being read from memory. That is, in general, for every cache line we bring in (which contains either 8 or 32 64-bit words), the processor only uses a few of them (the rest being useless traffic). This is no surprise, since both, *Nasa7* and *Arc2d*, have very large strides that cause this cache pollution. For *Swim256*, which is dominated by stride-1 memory accesses, the problem are cache conflicts. Although *Swim256* has almost no pollution (i.e, all data in a cache line will be used by the program), cache lines do not survive long enough in the data cache to be useful. They are evicted from the cache without using all the data elements. Then, these lines must be reloaded from main memory, thus increasing the main memory traffic.

We observe in figure 5.6 that the CB model generates less main memory traffic than the CA model. While the CA memory model reaches large values of non-filtered traffic, even

larger than 100%, the CB model reaches a maximum of 80% of non-filtered memory traffic for *Swim256* program. For the rest of the programs, the CB model achieves even lower values. The reason is the shorter number of L2 cache lines in the CA model, thus increasing the number of L2 cache conflicts and the main memory traffic.

Surprisingly, results in figure 5.6 show that, in terms of traffic, it is better to use 16-element vector registers with the CB model. The reason is that, as vector memory accesses move as much as 16 elements, a lower number of cache lines is needed to allocate the data, thus providing a lower probability of incurring in cache conflicts. However, we can not make any conclusion about the register vector length until performance results will be presented.

Multimedia programs are characterized by their low main memory traffic, which becomes negligible when the CB memory model is used, regardless of the vector length. These programs do not have very large working sets, so that most data fit in the cache hierarchy without being evicted due to conflicts. Therefore, data do not need to be reloaded from memory.

Overall, the important point is that, whether we choose the CA or CB models, a cache hierarchy can significantly filter processor traffic and thus lessen the pressure on the main memory system. Although the final performance measure will be EIPC, the traffic measure gives further information about which programs make the more intensive use of the main memory system. Given that each main memory access is expensive in terms of the number of cycles, it is very likely that the more main memory accesses the greater the number of cycles that the program execution takes. This fact will result in a small real performance, as we will see later. This kind of measures will allow us to explain the reason of the difference between ideal and real performance in the following sections and will also give us the support to propose solutions for enhancing real performance.

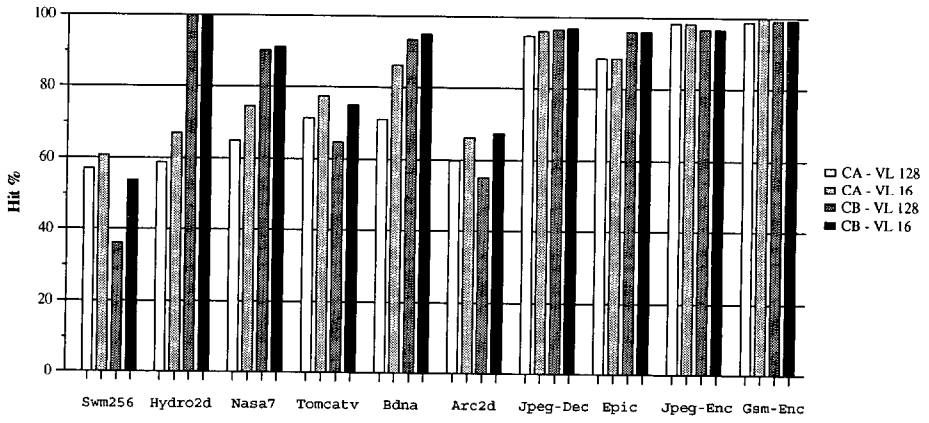


Figure 5.7 Hit percentage.

Cache Hit Rate

Although the traffic with the RDRAM has given some useful information about how the benchmark programs behave, we can not make a decision about which memory model will be better performance-wise only from the results in the previous subsection.

While the overall traffic filtering effect of CA and CB points to CB as the better memory model, the hit rate study can make us change our mind. The hit percentage metric presented in this section refers to the percentage of times that the processor makes an access to one data element and it hits. In the CA model, this value is the direct L1 hit rate. In the CB model, however, as the processor can directly access the L1 cache and the L2 vector cache, the measure must be calculated taking into account the number of elements that were accessed to the L1 cache (and the number of hits) and those that were accessed to the L2 Vector Cache (and its number of hits).

Figure 5.7 shows the hit rate for both models, CA and CB, when 128-element and 16-element vector registers are used. Clearly, the hit rate is better in the CB memory model. This was expected since, in the CB model, all vector accesses are served by a 1MB vector cache, while in the CA model, vector memory accesses are served by a 32KB vector cache. Will this higher hit rate provide a better performance? The L2 vector cache is 4 cycles away from the processor, while L1 cache is just 1 cycle away.

A back-of-the-envelope calculation would tend to favor the CA model despite its lower hit rate.

However, two factors combine to make CB a better alternative cycle-wise. First, vector codes have some inherent latency tolerance. Therefore, the longer latency might become a non-issue if, indeed, the CB can provide a higher effective sustained bandwidth. Second, communication between the L1 and L2 caches due to L1 misses is expensive and cannot always be overlapped with other cache requests. For example, in the CA model, an unaligned vector miss to L1 will *require*, by definition, two different lines from L2, therefore, at the very least, bandwidth is reduced to a half. Therefore, the CA model, where we have many L1 misses, delivers a lower effective bandwidth than CB.

In figure 5.7, we can also observe that using 16-element vector registers provides a higher hit rate in both, CA and CB memory models. The reason is the same as in the previous section, that is, using 16-element vector registers requires using less cache lines simultaneously, thus decreasing the number of conflicts in caches. Therefore, evicting less cache lines allows to hit much more often the next time that the elements of a certain line are accessed.

As a summary, we can say that the study of cache hierarchy efficiency shows that the CB model seems to be a better memory model, as it posses a lower pressure in the main memory and it hits much more often in caches. The 16-element vector registers also provide better results in terms of traffic with the main memory and cache hit rate.

5.4.4 Vector Cache Stall Time and Bottlenecks

Once we have studied the cache efficiency of the memory models in terms of traffic and hit rate, we will now analyze the characteristics of the memory hierarchy that are becoming performance bottlenecks for the execution of the benchmarks in both models. We will do it by studying the stall time in the vector cache as well as the reason of the stall. We have measured four different stall time reasons:

- Full Miss Status Holding Registers (MSHR). It happens when the number of MSHR entries is not enough to preserve the non-blocking nature of the cache. Recall from section 4.3.1, page 116, that 128-element stride-1 vector load may cause eight or nine cache line misses, thus requiring a large number of MSHRs.
- Full Write Buffers. It happens when the number of write buffer entries is not large enough to compensate the different speeds at which the processor generates store operations and the memory performs them.
- Coherency. The CB memory model presents coherency problems between the L1 and L2 caches because of the direct path between the processor and the vector cache. Solving these problems can stall the vector cache.
- Conflicts. Cache conflicts must be solved by ejecting the old cache line, and loading the new one. These situations can also stall the vector cache.

In the following sections we will present the total vector cache stall time, for the different memory models and vector lengths. Then, we will separately study the different stall time reasons.

Total Vector Cache Stall Time

Figure 5.8 shows the percentage of the total execution time that the vector cache is stalled for the CA and CB memory models and for 128-element and 16-element vector registers. In this figure, we can clearly observe that, in multimedia programs, the vector cache is hardly stalled during the program execution. These results were expected since we have seen in the previous sections that these programs have low traffic with the main memory and a high hit rate. By contrast, in numerical programs, the vector cache is stalled for many cycles during execution. We can see in this figure that fixing the vector length, starting from the 1x2 configuration and moving to the right, the CA memory model presents a higher stall time percentage. The exception are programs *Swim256*, *Hydro2d*, *Tomcatv* and *Bdna*. For these programs, the same happens up to the 4x8 configuration. However, at this point, the CB memory model stalls the vector cache for a higher percentage of the execution time.

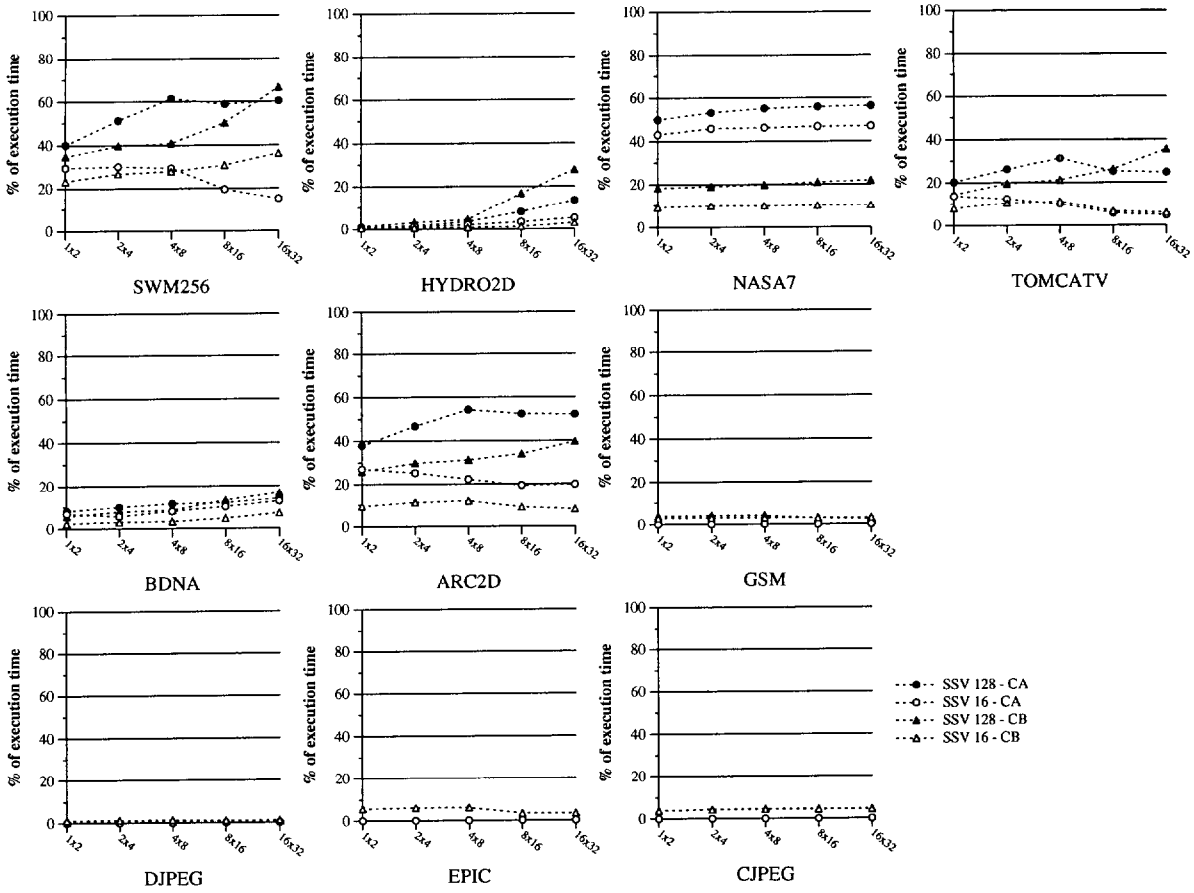


Figure 5.8 Percentage of the total execution time that the vector cache is stalled, for the CA and CB memory models, for vector lengths 128 and 16.

We can also observe in figure 5.8 that using 16-element vector registers stalls the vector cache for less cycles than using 128-element vector registers. Although we might think that it would be better to use 16-element vector registers, we can not forget the results discussed in section 3.8, page 77, which shown that the shorter the vector length, the larger the amount of instructions and operations executed. Moreover, we have also discussed in section 5.3 that the ideal performance results for 16-element vector registers are lower than for 128-element. Therefore, although the 16-element vector registers execution stalls the processor for less cycles, the question is: will the additional instructions and operations that must be executed increase the total number of execution cycles, thus reducing performance? This question will be answered in the following sections and it will allow us to make a decision about if it is better to use 16-element of 128-element vector registers.

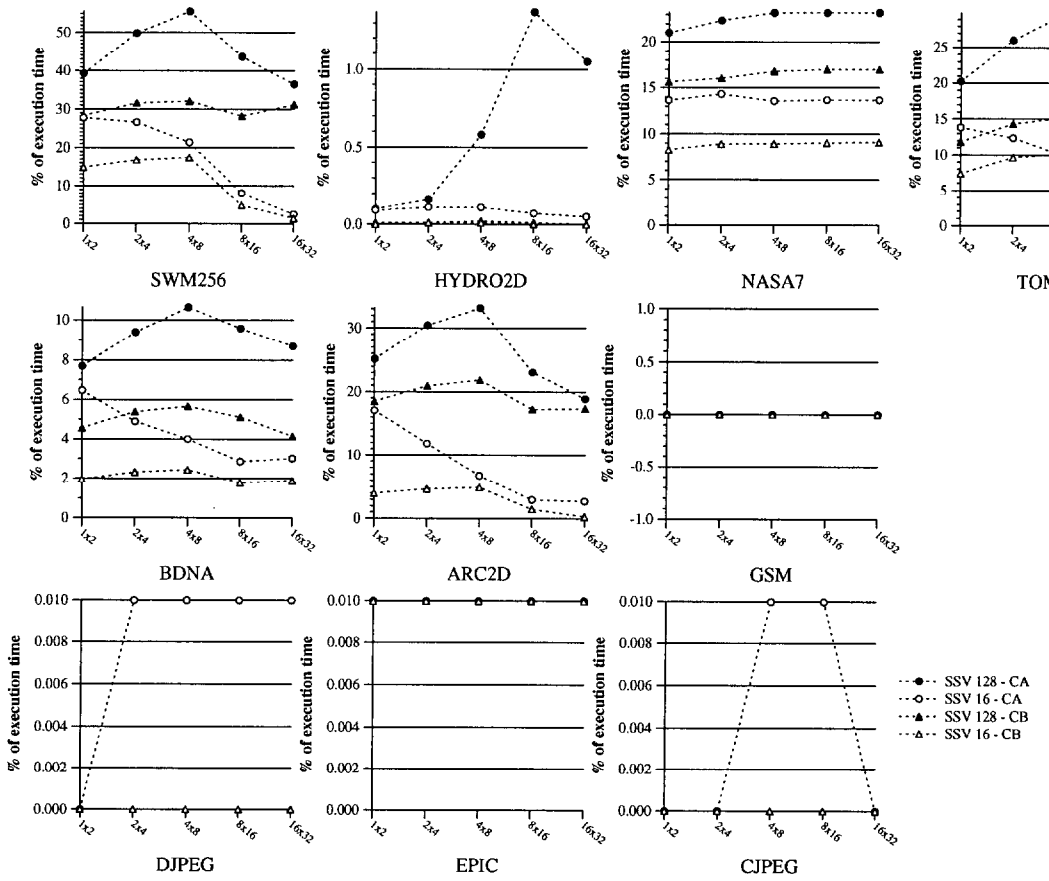


Figure 5.9 Percentage of the total execution time that the vector cache is stalled due to the Miss Status Holding Registers being full, for the CA and CB memory models, for vector lengths 128 and 16.

In the following sections, we will analyze the different reasons of the vector cache stalls in order to understand the memory bottlenecks that can potentially constrain the performance.

Vector Cache Stall Time due to full Miss Status Holding Registers

The first reason that can make the vector cache stall is the size of the MSHR. As stated in table 5.3, page 156, both, the CA and CB memory models, have 8 MSHRs. Each vector load instruction needs as many MSHRs as half the number of cache lines involved in the access. Thus, a 128-element vector load needs four or five MSHR entries

(2 lines/MSHRentry × 16 elements/line × 4 MSHRentries = 128 elements), depending on alignment, and a 16-element vector load instruction needs one MSHR entry. Thus, pressure on the eight MSHR entries available in CA and CB will be high.

Figure 5.9 shows the percentage of the total execution time that the vector cache is stalled due to the MSHR being full. The results in that figure correspond to the CA and CB memory models with 128-element and 16-element vector registers.

As we observe in that figure, the percentage of execution time that the vector cache is stalled due to the MSHR is negligible for multimedia programs. For numerical programs, however, it is rather important, being as large as 55%, 33% and 30% for *Swim256*, *Arc2d* and *Tomcatv*, respectively. Programs *Bdna* and *Nasa7* reach 13% and 23% of stall time, respectively; *Hydro2d* is the exception with only 1.4% of stall time due to full MSHR.

We additionally identify two behaviors in figure 5.9. On one hand, the 128-element vector register configurations stall the vector cache longer than the 16-element ones. The reason is that, with the same amount of MSHR in both cases, the 128-element registers need much more MSHR per access, so that they will fill up sooner, stalling the vector cache until they can be reused.

On the other hand, the CA memory model stalls the vector cache longer than the CB memory model. This is due to the shorter size of the vector cache in the CA memory model, which requires reloading evicted data, thus making an intensive use of the MSHR and increasing the stall time. Moreover, in the CB memory model, scalar accesses use the L1 cache MSHR, while vector access use the L2 cache MSHR, thus reducing the pressure on the vector cache MSHR.

From figure 5.9, we can conclude that the number of MSHR entries is a key factor of the memory model configuration as it can stall the vector cache up to 55% of the total execution time. Enough MSHR entries must be provided in order to decrease this stall time so that the memory hierarchy is not under pressure and a better memory behavior can be achieved.

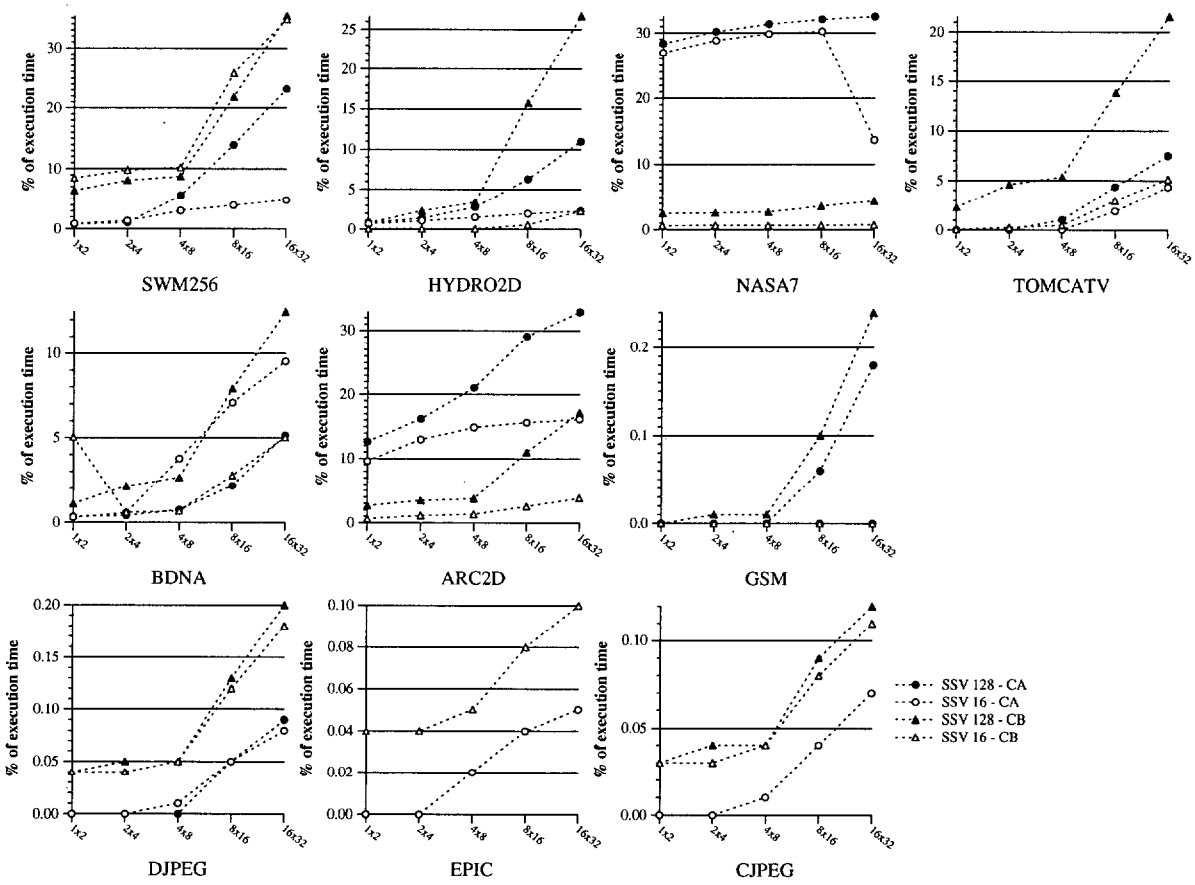


Figure 5.10 Percentage of the total execution time that the vector cache is stalled due to the Write Buffer being full, for the CA and CB memory models, for vector lengths 128 and 16.

Vector Cache Stall Time due to full Write Buffer

In the same way as the MSHR can stall the vector cache for many cycles, the Write buffers can also be an important stall reason. Every vector store needs as many WB entries as the number of cache lines involved in the access: eight or nine entries for the 128-element vector accesses (1 line/WBentry $\times$ 16 elements/line $\times$ 8 WBentries = 128 elements), depending on alignment, and one or two entries for the 16-element vector accesses, also depending on alignment. A high pressure on the write buffer can stall the vector cache for many cycles, which can indeed influence the performance.

Figure 5.10 shows the percentage of the total execution time that the vector cache is stalled due to the WB being full, for the CA and CB memory models with 128-element and 16-element vector registers. As expected, in multimedia programs the vector cache is stalled during a very small percentage of the execution time due to the WB being full. Numerical programs, however, stall the vector cache up to 35%, 33% and 34% for *Swim256*, *Nasa7* and *Arc2d*, respectively. Contrary to the results in the MSHR study, in this case, there is not a clear behavior about the memory model that stalls the vector cache for a longer time. In three programs, *Swim256*, *Hydro2d* and *Tomcatv*, the CB model stalls the vector cache for a larger percentage of the execution time. On the contrary, in *Nasa7* and *Arc2d* it is the CA model what stalls the vector cache during a longer time. In *Bdna*, it depends on the configuration and in multimedia programs the differences are so slight that we can not extract any strong conclusion. The same behavior appears for the different lengths of the vector registers.

What is clearly observed in figure 5.10 is that, as the configurations are scaled, the pressure on the WB increases, so that the vector cache is stalled for a larger amount of cycles. It seems that, as the processor configuration is scaled, the number of WB entries must also be increased.

Vector Cache Stall Time due to cache coherency

We now turn to the study of another reason that may stall the vector cache, that is, solving coherency problems between the L1 and L2 caches. As discussed in section 4.3.2, page 123, the direct path from the processor to the L2 vector cache can make coherency problems appear when some data are accessed in both, scalar and vector modes. When a L2 data is accessed through this direct path for writing, it may happen that a copy of the same data can be located in the L1 cache. Updating the L2 data will make L1 and L2 caches non coherent. As we have chosen to preserve coherence between both cache levels, we must invalidate the L1 copy of the data and update the L2 data. Of course, those invalidations consume some L2 cycles, thus preventing L2 from performing any other task. These coherency problems can only appear in the CB memory model due to the direct path to the L2 vector cache. The CA memory model does not spend any

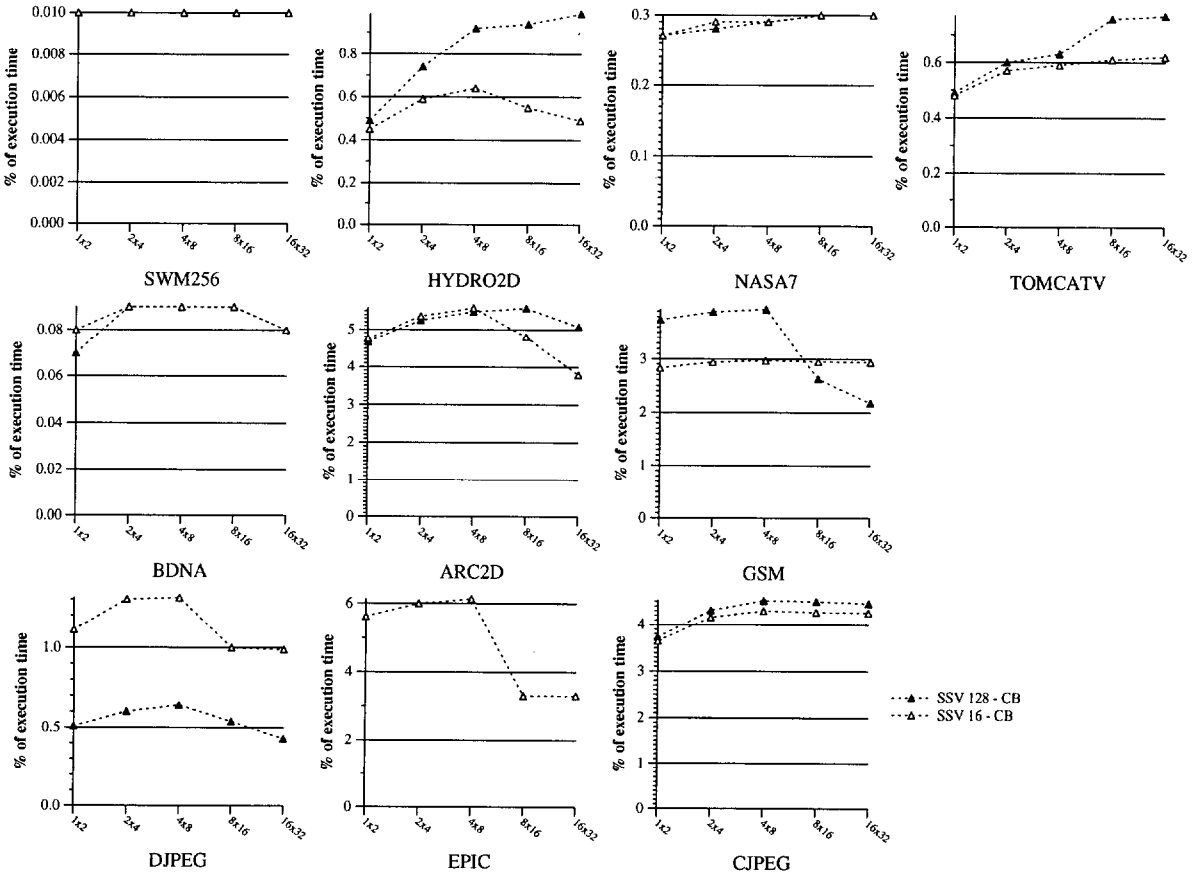


Figure 5.11 Percentage of the total execution time that the vector cache is stalled due to coherence problems for the CB memory model, for vector lengths 128 and 16.

cycle in maintaining coherency between caches, as data can only be accessed through the L1 data cache.

Figure 5.11 presents the percentage of the total execution time that the vector cache is stalled due to coherency problems for the CB memory model with 128-element and 16-element vector registers. We observe that, in general, the vector cache is stalled during a small percentage of the execution time for the majority of the programs, being *Arc2d*, *Epic* and *Jpeg Encode* the most affected programs, with stall time percentages of 6%, 6% and 4.6%, respectively. In general the 16-element vector registers provide a lower stall time percentage, except for *Jpeg Decode*.

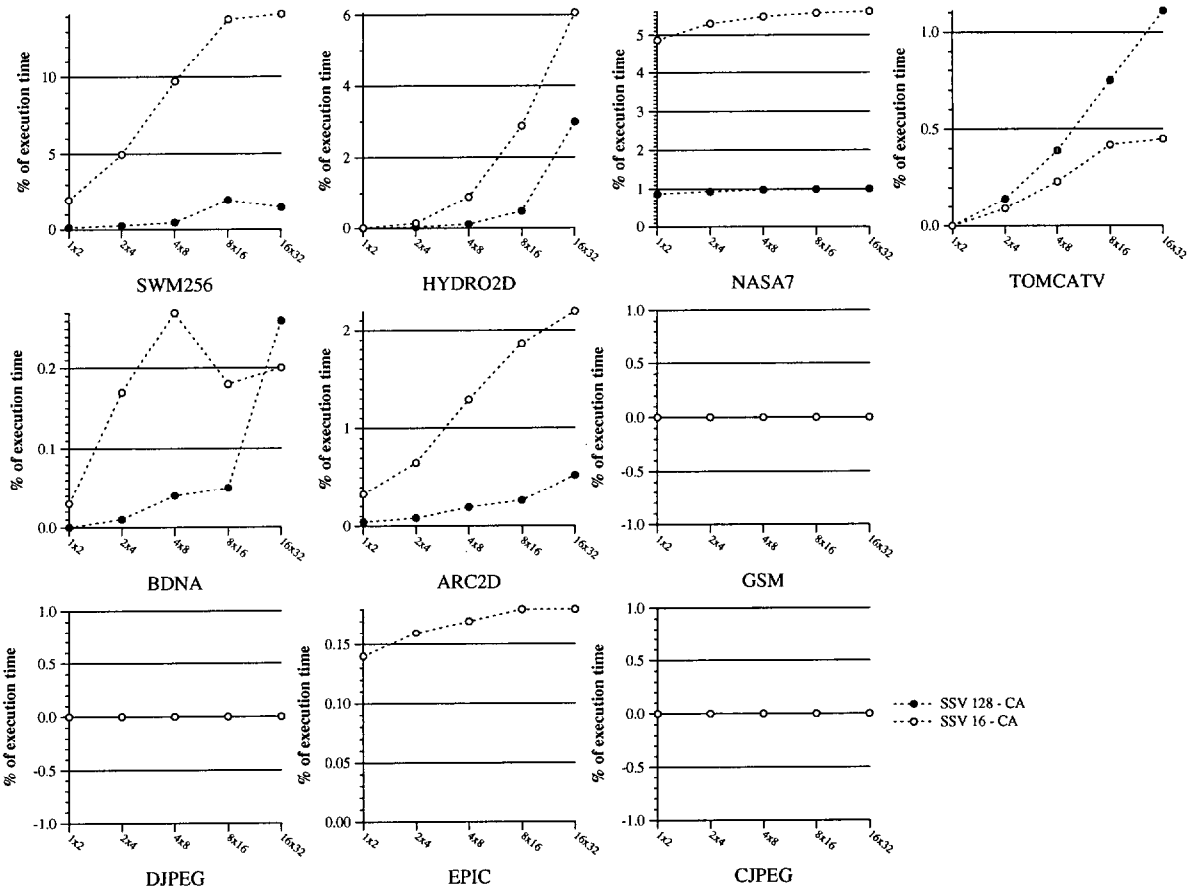


Figure 5.12 Percentage of the total execution time that the vector cache is stalled due to cache conflicts for the CA memory model, for vector lengths 128 and 16.

As we can observe in figure 5.11, the coherency maintainance is not an important stall reason when compared to the previous MSHR and WB studies. Thus, we will propose some enhancements to deal with the MSHR and WB stall time in the next chapter, but we will not address the coherency stall time.

Vector Cache Stall Time due to cache conflicts

Finally, in this section we present the study of the vector cache stall time from the cache conflicts point of view. Whenever a cache conflict ocurrs, one cache line must be evicted from the cache (and written to the upper memory level if necessary), and another line

must be loaded into the cache. All these tasks consume some time during which the vector cache is stalled and it can not work on another pending memory access.

The number of cache conflicts is related to the size of the cache and the number of cache lines. Therefore, it is very likely for the number of cache conflicts to be much higher in the vector cache of the CA memory model, since its size is much smaller than in the CB memory model. In this model, by contrary, the number of cache conflicts is very small because of the large 1 MB secondary cache, thus stalling the vector cache for a negligible amount of cycles.

Figure 5.12 shows the percentage of the execution time that the vector cache is stalled due to cache conflicts for the CA memory model, for 128-element and 16-element vector registers. We can see in that figure that the percentage of the execution time devoted to solve cache conflicts is as large as 14% for *Swim256*, 6% for *Hydro2d*, 5.6% for *Nasa7* and 2.2% for *Arc2d*. The rest of the programs only dedicate between 0% and 1% of the total execution time. The reason for this behavior is the size of the program working set, which is larger in the first set of programs.

Summing up, there are four reasons for the stall of the vector cache, that is, full MSHR, full WB, coherence problems and cache conflicts. While the first two reasons can stall the vector cache for many cycles, the other two only affect one of the memory models, and only to a shorter extent. In general, numerical programs are affected by bottlenecks related to their large working sets. They make a higher use of the main memory bus, causing stalls in the vector cache when the MSHR or the write buffer entries fill up. Clearly, the programs that put most pressure onto the RDRAM array are the ones that stall the most due to MSHR being full. In total, the vector cache can be stalled up to 60% of the total execution time. This fact will influence the execution time of the different programs and the performance results that will be studied in the following sections.

5.4.5 Performance Evaluation

In this section, we will study the CA and CB memory models from the performance point of view. Although the studies of cache hierarchy efficiency and the stall time provide with useful information about the behavior of both models when 128-element and 16-element vector registers are used, the performance study carried out in this section will give us information about which memory model carries out a faster execution of the benchmark programs. Moreover, in this section, we will also compare the performance behavior with a typical SS architecture backed with a cache hierarchy. This comparison allows us to determine whether the SSV architecture is worth at the performance level, when a real cache hierarchy is attached.

General Performance

Figure 5.13 compares the performance of the two cache models, CA and CB, both with 128-element and 16-element vector registers, against the performance of the SS machine with a real cache memory system.

From results in figure 5.13, we can see that, in general, the SSV architecture, both with CA and CB memory models, reaches much better performance than the SS architecture. Regarding the memory models of the SSV architecture, for the numerical benchmarks, the CB model outperforms the CA model, the 128-element vector registers also provide better performance results. This is especially true for the aggressive configurations. The reason is that these programs are highly vectorizable and they use large vector data, so that, the larger the vector length, the shorter the number of instructions and operations executed, and the better the performance. This is the case of programs *Swim256*, *Hydro2d*, *Tomcatv*, *Bdna* and *Arc2d*. For the *Nasa7* program, the CB memory model also achieves a better performance than the CA memory model. In the CB memory model, the 128-element vector registers reach higher performance values. In this program, the CA memory model provides low performance results, as even the SS architecture outperforms its EIPC values.

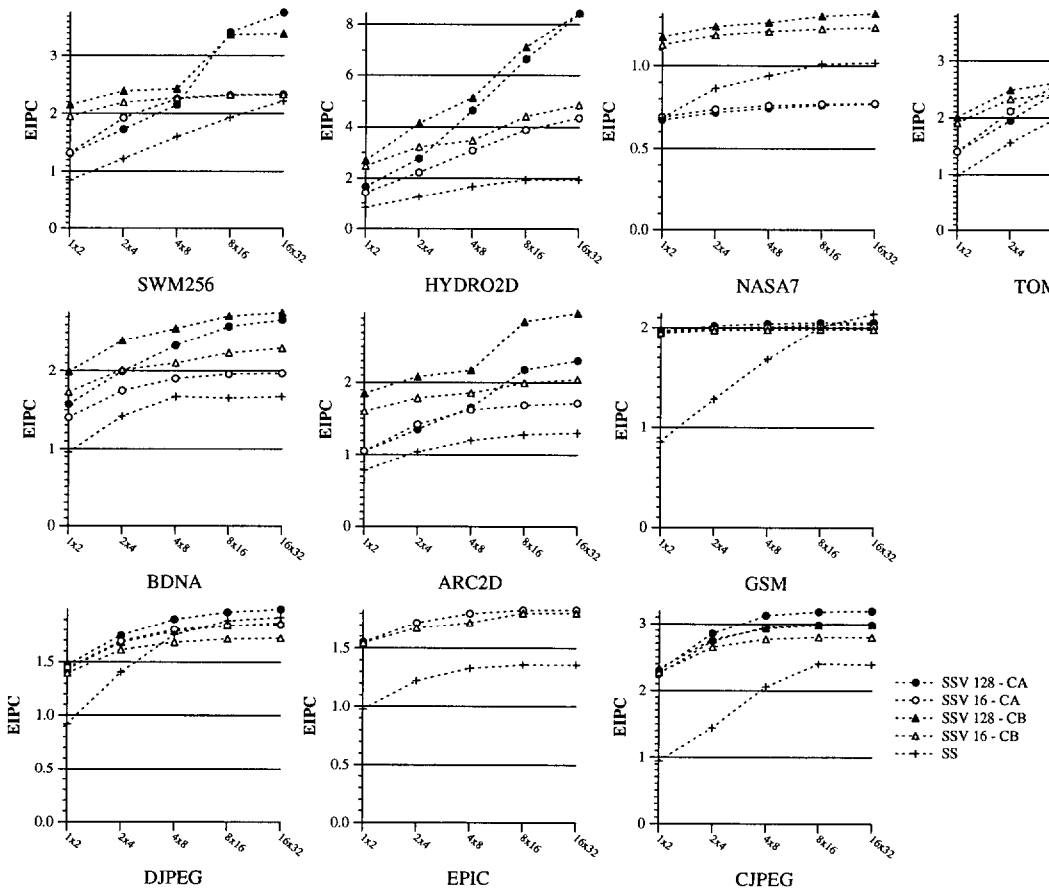


Figure 5.13 Performance evaluation of the SSV architecture backed with the CA and CB memory models, for vector lengths 128 and 16, and comparison with the SS architecture backed with a typical cache hierarchy.

On the other hand, for the multimedia benchmarks (*Gsm Encode*, *Jpeg Decode*, *Epic* and *Jpeg Encode*), although the CA model outperforms the CB model, the difference in performance results is quite small. The CA model works a little bit better because of the high number of reduction operations (such as adding all elements of a vector), which are very common in image applications. The results of these reductions are individually written to their respective matrix positions using a scalar store. Therefore, in the CB model, these data items go to the L1 cache. Later, the vector unit requests a bunch of these results in vector mode to the L2, only to find that there is a coherency problem with L1. The L2 must flush and invalidate data from the L1 to service the vector request, resulting in a loss of performance.

Comparing the results of the real cache models, shown in figure 5.13, with the results for an ideal memory system, already shown in figure 5.1, page 142, we can observe that, in general, the multimedia programs reach real cache performance results very close to the ideal memory performance results. Numerical programs, however, present a substantial performance loss when a real memory system is attached to the SSV processor. Among the numerical programs, *Swim256*, *Nasa7*, *Arc2d* and *Tomcatv* suffer higher performance loss (99%, 75%, 95% and 60% of performance loss relative to the ideal EIPC, respectively). *Swim256* has a large working set that does not fit in the cache hierarchy, so that conflicts evict useful data from the cache and these data must be reloaded later. *Tomcatv* presents a similar behavior. *Nasa7* and *Arc2d* are characterized by non stride-1 vector memory accesses. The performance results of *Nasa7* are aggravated by the fact that only 35.6% of the accesses are stride-1 (i. e. only 35.6% of the accesses can be served at the maximum port bandwidth). In fact, the same case happens to *Jpeg Encode*, where the CB results are close to the ideal ones for conservative configurations but scale poorly as we simulate more aggressive configurations. Again, the explanation lies on the low percentage of the vector accesses with stride-1. Note that the problem of non stride-1 accesses could be solved in some cases with a cautious code rewriting. For example, *Jpeg Encode* and *Jpeg Decode* are characterized by having a structure where the three color components are interleaved. This results in stride-3 memory accesses when performing the color conversion algorithm. The code could be easily rewritten to change these accesses to stride-1. All these considerations must be added to the vector cache stall study carried out in the previous sections.

In short, we can conclude that numerical programs are limited by the memory system performance, while multimedia programs are not. In fact, these results match the memory traffic and hit rate results analyzed in the previous section. In that section, we could see that multimedia programs had a very low main memory traffic, especially when the CB memory model is used, and their hit rate was higher than 90%. Therefore, it seems that the main constraints appear in the execution of the numerical programs, and the bottlenecks will be located in the memory hierarchy. These programs did not fit in cache, their hit rate was very low in some cases (see again figure 5.7, page 162), and the main memory traffic was an important fraction of the words that the processor accessed (as seen in figure 5.6, page 160).

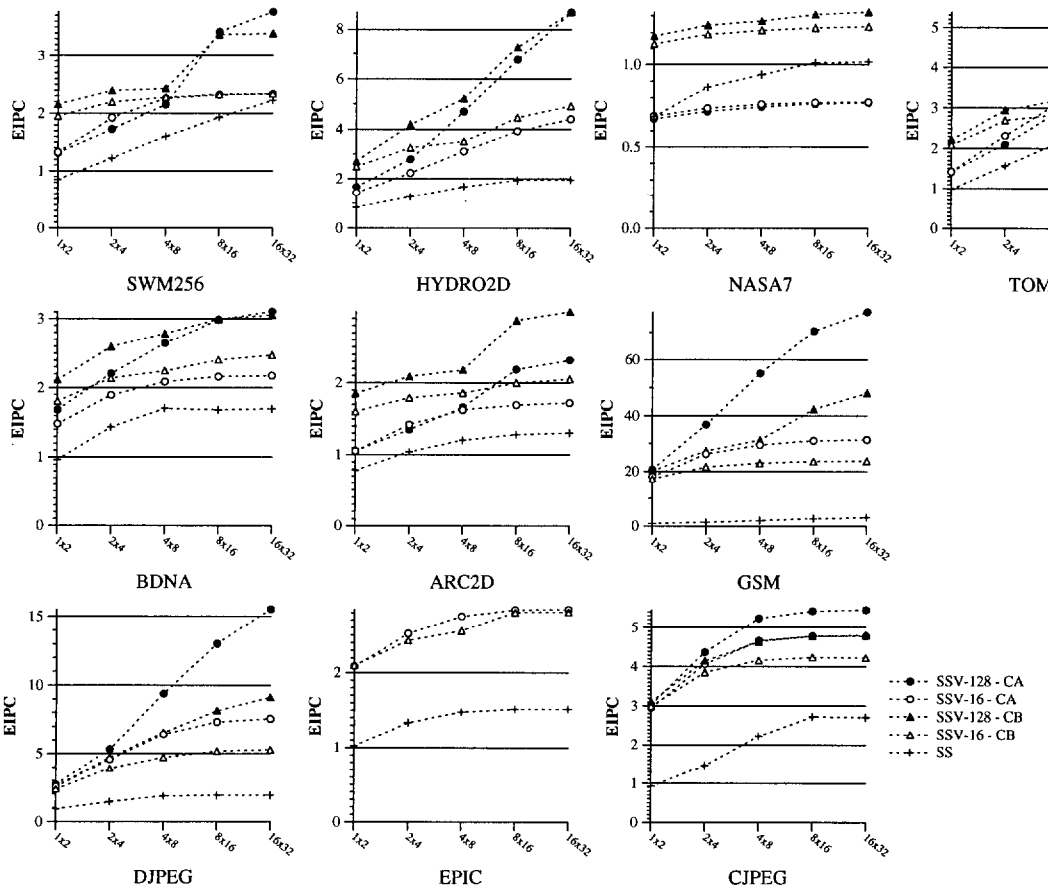


Figure 5.14 Performance results inside D-regions for the SSV architecture backed with a real memory (CA and CB memory models), and for the SS architecture backed with a typical cache hierarchy.

Performance Breakdown by Regions

As in the previous sections, we will now analyze the performance results inside D-regions and S-regions for the different configurations of the SSV architecture and memory models. We will also compare these results with the SS architecture performance results inside D- and S-regions, respectively.

Figure 5.14 shows the performance results inside D-regions for the SSV architecture (backed with the CA and CB memory models) and for the SS architecture (backed with a typical cache hierarchy). Comparing these results to those presented in figure 5.13, we can observe two different behaviors. First, for most of the programs, the results shown

in this figure follow the same trends than those presented in figure 5.13, meaning that the overall performance is mainly determined by the performance obtained inside D-regions. In some of these programs, the reached values are higher than the same values in the overall performance. The overall performance results are lower because they include the relatively low EIPC values reached inside S-regions. The second behavior appears in *Gsm Encode* and *Jpeg Decode* programs. For these programs, we observe that, while their overall performance results flattened as the machine configuration was scaled, their performance results inside D-regions scale quite well with the processor scaling. Therefore, the performance flattening effect is caused by the behavior inside S-regions and their weighted contribution to the overall performance. This behavior appears in low vectorizable programs with pure D-regions, that is, D-regions including few scalar instructions. Moreover, as these two programs are not limited by the memory behavior, they can expose a large amount of parallelism that is easily exploited by the SSV architecture.

It is important to note here that the spectacular EIPC values for the *Gsm Encode* program appear because of the privatization of the data structures that we made in order to vectorize the program. These data privatizations reduced the number of load and store instructions executed by the program, which in turn reduced a lot the number of execution cycles. Although in the scalar version of the program we made the same changes, the compiler was not able to reduce the number of instructions executed. Therefore, whenever we calculate the EIPC measure we make the relation between a large amount of scalar instruction and a short amount of cycles, thus generating large EIPC values. These large results do not appear in the overall performance because *Gsm Encode* presents a low vectorization percentage so that the overall performance results come mainly determined by the behavior inside S-regions.

As stated for the overall performance, for numerical programs, the CB memory model achieves a better performance, while for multimedia programs the CA memory model presents higher results. In general, using 128-element vector registers provides a higher performance, which was expected since less instructions and operations are executed. Regarding the comparison to the SS architecture, the SSV architecture reaches a better performance, regardless of the memory model, either CA or CB.

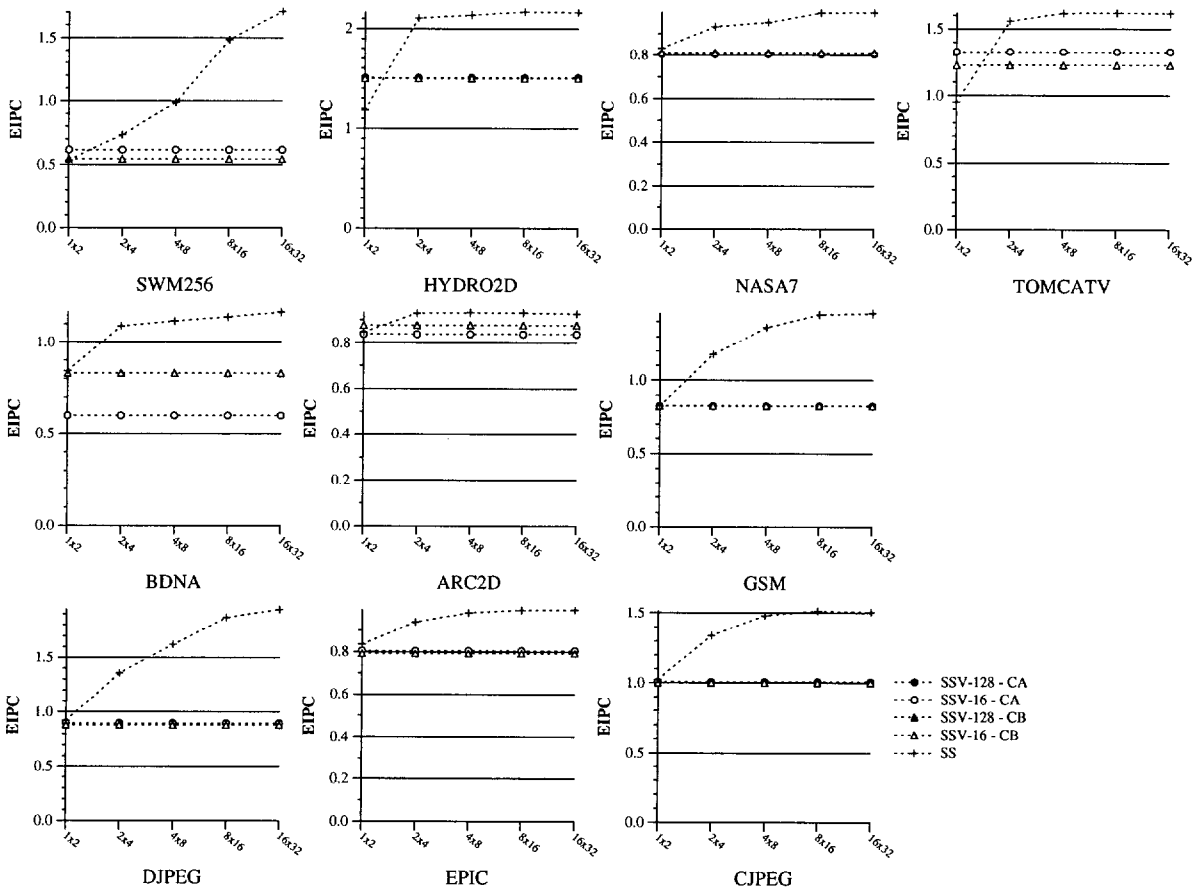


Figure 5.15 Performance results inside S-regions for the SSV architecture backed with a real memory (CA and CB memory models), and for the SS architecture backed with a typical cache hierarchy.

Figure 5.15 shows the performance results inside S-regions for the SSV and the SS architectures, with real memories attached. In general, the trends in this figure are the same as those already studied in figure 5.3, page 148, where the performance inside S-regions for an ideal memory system was presented. As occurred in that case, starting in the 1x2 configuration, the SSV and SS performance results are very similar. As configurations are scaled, however, the SS performance results are clearly better than the SSV results. All in all, the achieved performance values are quite low and the SS performance flattens rather quickly. Therefore, as concluded before, although a certain out-of-order execution is needed, the more aggressive superscalar core does not provide with a substantially better performance. Instead of investing the transistor budget in scaling the superscalar core, once the performance results flatten, the addition of vector

capabilities would be helpful in order to exploit more parallelism during programs execution.

Note that, in figure 5.15, the behavior of the *Swim256* program for the SS architecture does not follow the behavior previously described. As discussed in section 3.7, page 67, this program is 99% vectorizable, it hardly executes 0.01% of its total number of operations inside S-regions and the average size of the S-regions, in operations, is 21. It means that with such small S-regions, the behavior inside S-regions will be affected by the overlapping between consecutive D- and S-regions. As explained and discussed in section 2.6.3, page 48, some frontier instructions, which belong to a certain region (either S- or D-), can be effectively executed (and accounted for EIPC purposes) in the consecutive region. In general, this overlapping is very low. However, when the size of the S-regions is so short, as it is in the *Swim256* program, these overlapping operations can pollute the EIPC measure and reduce its meaningfulness. As these extra operations are added to the region's operations and no extra cycles are included, the EIPC measure grows significantly. As the processor configuration is scaled, the higher aggressiveness of the issue engine drives to have more instructions executing on-the-fly, and the overlapping effect also increases. Whenever we have large S-regions, these extra operations can certainly pollute performance, but in a very slight amount. For that reason, the *Swim256* program presents such spectacular increase in performance inside S-regions.

5.4.6 Data Parallelism Inside Vector Regions

We will now present the amount of data level parallelism that the SSV and SS architectures extract inside D-regions when a real memory hierarchy is added. Similarly to section 5.3.3, we have computed the "operations per cycle" rate (OPC) within each D-region. Figure 5.16 compares the distribution of OPC for the SSV and SS architectures for the lower and upper bound of the configuration set, that is, 1x2 and 16x32. It also shows the OPC for 128-element and 16-element vector registers.

As stated in section 5.3.3, each bar in figure 5.16 shows the percentage of the total number of operations that are executed at a certain OPC.

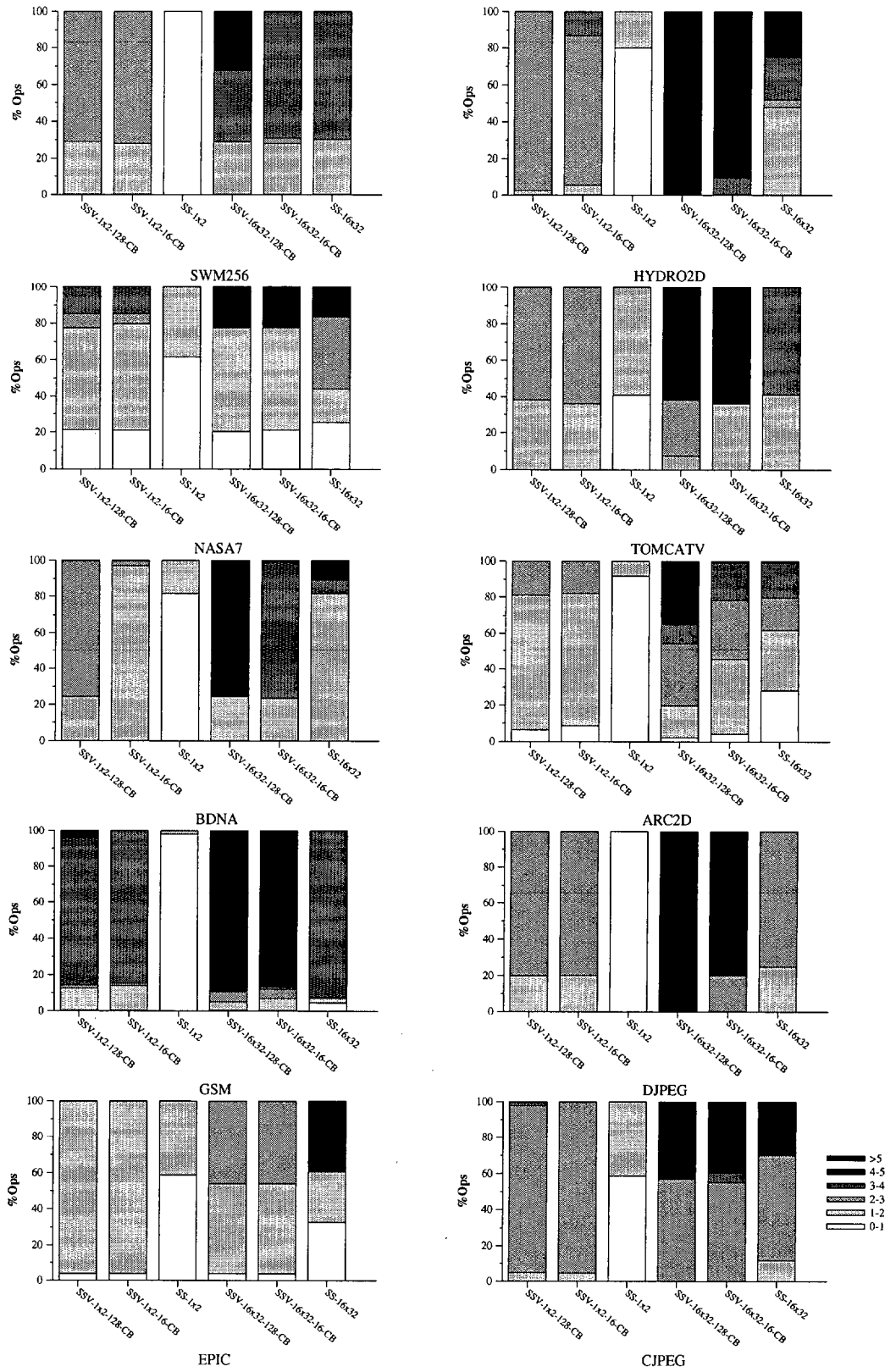


Figure 5.16 Distribution of operations percentage executed within a certain OPC range, inside D-regions.

Comparing figures 5.4 and 5.16, we observe that the addition of a real memory hierarchy to both, the SSV and SS architectures, diminishes significantly the amount of data level parallelism that the architectures are able to exploit. This is especially true for the SS architecture as, except for *Jpeg Encode*, it does not execute operations with OPC larger than 5 when the real memory hierarchy is introduced.

In figure 5.16, we observe that, for the SSV architecture, the obtained OPC increases when the processor configuration is scaled. These increases are more important for some programs, like *Hydro2d*, *Gsm Encode*, *Tomcatv* and *Jpeg Decode*, which in the 16x32 configuration execute 99%, 90%, 60% and 80% and of their operations, respectively, with OPC larger than 5.

Some of the benchmarks, like *Swim256*, *Hydro2d*, *Bdna*, *Arc2d* and *Gsm Encode*, change their behavior depending on the vector length. These programs exploit more data level parallelism with 128-element vector registers, as was expected.

Regarding the comparison with the SS architecture, the general trends remain the same as those already commented in section 5.3.3. The SSV architecture is able to exploit much more data level parallelism than the SS architecture, reaching, for seven programs, OPC larger than 5.

As a conclusion of the real cache study, we have seen that the SSV architecture is a competitive design that reaches better performance results than the SS architecture with ideal and real memory systems. Regarding the two cache models connected to the SSV architecture, the CB model presents a lower main memory traffic for both, numerical and multimedia programs. It also presents a higher hit percentage for numerical programs, which are more limited by memory than the multimedia programs. Regarding the performance study, numerical programs reach better performance values when the CB memory model is used. Although multimedia programs achieve higher performance for the CA memory model, the difference with the CB performance results are small. These results, together with the potential disadvantages of the CA model already discussed in section 4.3.2, page 123, present the CB model as the selected memory hierarchy for the SSV architecture that we propose. The rest of this work will

use the CB memory model as the cache hierarchy to which the SSV architecture is connected.

In these sections, we have also detected that there is an important performance loss in numerical applications when changing from an ideal to a real memory hierarchy. Some of the numerical programs are limited by the memory behavior, as we have studied in the previous sections. In the following chapter, we will try to decrease the memory pressure by tuning the memory hierarchy design.

5.5 SUMMARY

In this chapter we have presented the performance results of our proposed superscalar architecture with a vector unit and we have compared it with a traditional superscalar processor. We have studied two different aspects: on one hand, we have studied the scalability and potential performance of the SSV and SS architectures when a perfect memory is used. We have observed that the SSV architecture scales very well as more memory and computing resources are added to the processor. Moreover, it reaches higher values of parallelism than the SS architecture with a lower cost and control complexity.

The analysis inside D-regions and S-regions has shown that the SSV architecture achieves high values of parallelism inside D-regions. However its contribution to the overall performance is determined by the relative weight of D-regions in the whole programs. For highly vectorizable programs, the overall performance is mainly determined by the high performance values obtained inside D-regions. However, low vectorizable programs have an overall performance that is also determined by the performance inside S-regions. As was expected, the performance inside S-regions is better for the SS architecture. The reason is that the superscalar core of the SSV architecture remains constant along the different configurations, while, in the SS architecture, the core is scaled adding more and more resources.

The study of the amount of data parallelism that each architecture is able to reach inside D-regions has shown that, although both, the SSV and the SS architectures, obtain larger OPC values as the configurations are scaled, the SSV architecture achieves larger values than the SS architecture in all programs.

On the other hand, we have also studied the performance of the SSV when a real cache hierarchy is introduced. We have studied two different cache hierarchies, CA and CB, based on the introduction of the vector cache. CA and CB models differ on where the vector cache is located. While in the CA model the vector cache is in the first cache level, in the CB model, the vector cache is in the second cache level. We have also included the performance values that a traditional superscalar processor achieves when a typical cache hierarchy is introduced. As a first step, we have studied the cache hierarchy efficiency of the CA and CB models in terms of traffic with the main memory and hit/miss rate. We have observed that regardless of the memory model, CA or CB, the cache hierarchy is able to filter the processor's traffic. We have also observed that multimedia programs have a very low memory traffic, while numerical programs present a lower amount of memory traffic when the CB model is used. The results of the cache hit rate show that multimedia programs hit much more often in cache than numerical programs. Again, the CB model reaches higher hit rate percentages. Therefore, the cache efficiency study has shown that numerical programs make a higher pressure on the main memory, and that the CB model reacts much better to this pressure with lower main memory traffic and higher hit rates.

The study of the vector cache stall time has revealed that this cache can spend up to 60% of the total execution time stalled due to different reasons. The two main reasons are full MSHR and WB entries. The other two reasons (coherence and cache conflicts) contribute in a lower degree to that bottleneck.

The performance evaluation with a real cache hierarchy has shown again that numerical programs are limited by the memory system, while multimedia programs are not. Although the introduction of a real memory system has strongly reduced the performance values obtained by the SSV architecture, these performance results are still higher than the SS performance values. The study inside regions has presented that,

for the SSV architecture, programs scale well inside D-regions and behave constant inside S-regions. The overall performance is determined by the weight of the S- and D-regions in the whole program. Again, inside S-regions, the SS architecture reaches a better performance due to the scaling of the superscalar core.

The study of data level parallelism for a real memory has shown the same trends than for the ideal memory system, that is, the SSV architecture reaches higher OPC levels than the SS architecture and these values scale as the processor configuration is improved.

All in all, the CB memory model presents better performance results for the numerical programs and the difference with the CA performance results for multimedia programs is small. Therefore, the CB memory model will be the memory model that we propose to be attached to the SSV processor, as it achieves the better results both, in the cache efficiency and performance studies.

We conclude that the SSV architecture is a feasible architecture from the performance point of view. It reaches a better performance than a traditional SS architecture, either with ideal or real memory systems, as the processor configuration is scaled. It is a good choice for multimedia programs and it achieves really good results for numerical programs. These results can still be improved by tuning the memory hierarchy, as we will see in the following chapter.

Regarding the size of the vector length to be used, we consider that, although, in general, the better performance results are obtained by using 128-element vector registers, the final decision depends on the available transistor budget. An implementation for a general purpose processor should use 128-element vector registers in order to achieve the better performance. However, a low cost approach can sacrifice some of the obtained performance in order to decrease its cost, thus including, in such case, 16-element vector registers.

The performance results presented so far do not take into account cycle time. In the previous sections, we have discussed qualitatively the differences between the proposed

SSV architecture and a plain superscalar architecture. While both machines have the same basic resources (in terms of arithmetic functional units), Lee argues very convincingly in [LD97] that the vector datapath takes much fewer area and is much simpler (read, faster) than the equivalent superscalar datapath. Furthermore, the area devoted to control (fetch, decode, reorder buffer, TLB ports, etc.) in the basic superscalar processor grows quadratically as we scale the issue degree. Not only that, but as Palacharla et Al. [PJS97] argue, cycle time is also affected and does not scale as we increase issue width. Meanwhile, in the SSV machine, control logic has been kept exactly the same across all configurations. This is only possible thanks to the properties of vector instructions. Scaling the number of functional units *does not* require adding more issue slots or larger instruction queues.

It is also worth mentioning the huge difference in cache and TLB ports: the plain superscalar needs a 16-ported TLB and cache while, using the vector unit, we can get away just using a single TLB port and a single cache port. Everything considered, it is very likely that the SSV machine will have a faster cycle time than a hypothetical equivalent plain superscalar and, at the same time, the SSV will require less area.

6

IMPROVING PERFORMANCE BY TUNING THE MEMORY HIERARCHY

Summary

This chapter presents a study about tuning the memory hierarchy of the ILP+DLP architecture. The goal is to achieve a better performance by reducing the negative effect of the bottlenecks detected in the previous chapter. The tuning consists in improving the memory hierarchy by increasing non-blockingness, adding one extra scalar port and increasing the main memory bandwidth. We also study the effect in performance of the increasing technology integration scale by exploring the performance behavior as the size of the L2 data cache is increased. Finally, We evaluate two additional cache designs that try to alleviate the memory pressure of non stride-1 memory references.

6.1 INTRODUCTION

In the previous chapter we detected some bottlenecks related to the memory system configuration. We observed that some programs have large working sets, and because of the cache hierarchy organization these programs have a large amount of traffic with the main memory, and a relatively low hit rate. We also observed that the vector cache could be stalled during many execution cycles because of the filling up of the Miss Status Holding Registers (MSHR) and the Write Buffer (WB) devices. Maintaining coherence between the two cache levels and cache conflicts can also stall the vector cache. This chapter presents a study about the tuning of the SSV architecture in order to decrease the effect of these memory bottlenecks. The tuning consists in specific actions, such as increasing non-blockingness, adding one extra data port for scalar accesses, improving the memory bandwidth, and attacking the problem of strided accesses by evaluating performance under more sophisticated second level cache designs. Moreover, as technology evolves, more and more transistors will be integrated on a chip. Therefore, we will also look at the performance behavior as the size of the L2 data cache is increased. We will see that the performance of the SSV architecture can be substantially improved with these adjustments in the memory hierarchy design.

The tuning study will be carried out for a selected SSV configuration, out of those studied in the previous chapter. Among the different configuration parameters that must be fixed, it is necessary to select the processor configuration, as well as the memory hierarchy design and the size of the Vector Register File (that is, the length of each vector register). Let us discuss each parameter in detail:

- **Processor configuration.** As stated in the previous chapter, in general, the more aggressive the configuration, the higher the performance obtained. However, the hardware resources needed to build such aggressive configurations are far beyond today's implementation capabilities, so we must concentrate on the modest configurations in order to keep the study in the feasible range of SSV machines. We have also observed that the performance increase is larger in the smaller configurations, and this increase is comparatively shorter as the configurations are scaled. Therefore, following a cost/performance relation criterion, we must choose a con-

figuration that belongs to the low scale of the configuration set. In particular, we have selected the SSV-4x8 configuration, already defined in table 5.1, page 136. The major functional units of the SSV-4x8 configuration are a single memory port, two vector integer functional units and two vector floating point units. As described in section 4.3, the memory port can transfer four 64-bit words when operating in vector mode, but only a single 64-bit word when operating in scalar mode. The vector functional units can operate on multiple operands simultaneously. We have included two functional units, each one with four replicas (8 flops/cycle), in order to get a balanced configuration that matches the number of "lanes" in the vector unit (width of each vector unit) with the number of words per cycle read from the vector cache. This matching simplifies the steering logic that has to distribute each of the four words read from the cache to one of the four lanes.

- **Memory system configuration.** Among the two different proposed designs that have been evaluated in the previous chapter, CA and CB, we have selected the CB memory hierarchy model. We have seen in the previous chapter that the CB memory model obtains a better performance than the CA model for numerical programs, while the difference with the CA model for multimedia programs is quite small. The CB memory model also obtains better results in the cache efficiency study. Moreover, as we have discussed in the previous chapter, the CB memory model is a better alternative cycle-wise because of the inherent latency tolerance of vector code and the larger effective bandwidth that the CB model delivers.

  Additionally, using the CB model favors the execution of scalar pieces of code as it follows a memory model that is very similar to the classic memory hierarchy attached to a superscalar processor. For any scalar memory access the processor accesses the 1-cycle latency L1 data cache. Missing this cache implies accessing the L2 data cache, which has a higher latency and larger cache lines. The L1 and L2 data caches are connected by using a 4-word width path. The direct connection between the processor and the L2 cache, and the VRF, are not used, as no vector memory accesses are made.

  The basic configuration parameters of the CB model that will be used in this chapter are those already shown in table 5.3, page 156.

- **Size of the Vector Register File.** Finally, we must also fix the total size of the Vector Register File (VRF). Given that, as discussed in section 4.2, page 113, the appropriate number of logical and physical vector registers has been fixed to 16 and 32 registers, the only design parameter that can be set is the length of each individual vector register. As the aim of this study is the achievement of the better performance for the vector code execution, we have selected large vector registers (as shown in previous chapter, large vector registers provide higher performance results). Each individual vector register will have either 64 or 128 elements, which will imply a VRF size of 32KB or 64KB. We will present performance results for two different large vector lengths in order to study if there is a significant performance gain in using a double-sized VRF. If this is the case, the use of a greater VRF will be justified. Otherwise, the extra 32KB of chip space can be used in a different way in order to improve the overall performance.

We will compare the SSV performance results with a SS architecture. We will use the SS-2x4 superscalar configuration defined in table 5.1, page 136, backed with the typical cache hierarchy defined in table 5.3, page 156. As explained in section 5.2, this SS configuration consists of 2 independent memory ports (64-bit word each), 4 integer functional units and 4 floating point units. To issue enough instructions to all these units in parallel we need an 8-way superscalar core that is able to fetch, decode and commit 8 instructions per cycle.

The reader might complain that we are overpowering the SSV architecture (8 flops SSV-4x8 versus 4 flops SS-2x4), but, as discussed in the previous chapter, the eight flops of the SSV architecture come from only two different functional units executing two vector instructions in parallel. This design is simpler than the four functional units of the SS-2x4 architecture, which are independent of each other and, therefore, are expensive to implement. By selecting SSV-4x8 and SS-2x4 we have chosen two configurations with similar complexity. Moreover, it is precisely the point of this thesis that by using a vector unit the SSV-4x8 can achieve 8 flops per cycle without requiring a very-wide issue engine.

Another important difference between the SSV-4x8 and the SS-2x4 configurations appears in the four 64-bit words per cycle that the SSV-4x8 architecture is able to transfer in stride-1 vector accesses, while the superscalar architecture can transfer just two 64-bit words per cycle. The single port of the ILP+DLP is very simple to implement, while implementing more than two or three ports in a cache is not feasible with true multi-ported caches, and alternative designs using multiple banks and hybrids of multi-bank and multi-port must be used [JNT97] [RTAD97].

Perhaps, the reader might have expected a comparison between the SSV-4x8 and SS-4x8 configurations. However, as discussed in section 5.2, page 135, two equally named configurations differ significantly in complexity terms. Although they have the same total memory and computing power, the SS architecture is more complex than the SSV architecture because of:

- **the independent memory ports**: four different memory ports in the SS architecture versus one simple memory port in the SSV architecture.
- **the independent integer and floating point functional units**: four independent functional units in the SS architecture versus two independent functional units in the SSV architecture.
- **the wider superscalar core**: A 16-way out-of-order core in the SS architecture versus a 4-way out-of-order core in the SSV architecture.
- **the larger reorder buffer**: A 128-entry reorder buffer (and its more complex wake-up logic) in the SS architecture versus a 64-entry reorder buffer in the SSV architecture.
- **the larger Load/Store queue**: A 64-entry load/store queue in the SS architecture versus a 32-entry in the SSV architecture.
- **the improved branch predictor**: A 2K BTB and 16K Lbranch two-level adaptative branch predictor in the SS architecture versus a 0.5K BTB and 4K Lbranch 2-level adaptative branch predictor in the SSV architecture.

Therefore, fixing the budget that would allow us to implement the SS-4x8 architecture, we could make a more complex SSV counterpart. In other words, fixing the SSV-4x8 budget, the SS machine that matches the same complexity is not the SS-4x8, but the SS-2x4 configuration.

Once we have defined the configuration parameters of the architecture, in the following sections we introduce and evaluate some architectural improvements that can be included in the memory system of the basic SSV architecture, based on the performance bottlenecks detected in the previous chapter.

6.2 INCREASING NON-BLOCKINGNESS

In section 5.4.4, page 163 in the previous chapter, we observed that the vector cache was stalled for many cycles due to the Miss Status Holding Registers (MSHR) and Write Buffers (WB) being full. Vector accesses consume a large number of entries in these data structures so that when they fill up, they stall the processor for many cycles. Table 6.1 shows a summary of the percentage of time that the vector cache is stalled and the reason of the stall for the SSV machine configuration under study. These data have been extracted from figures 5.8, 5.9, 5.10 and 5.11 in the previous chapter.

	Total	Breakdown by stall reason		
		MSHR Full	WB Full	Coher.
Swim256	40.83	32.08	8.74	0.01
Hydro2d	4.35	0.02	3.41	0.92
Nasa7	19.82	16.80	2.73	0.29
Tomcatv	21.13	15.13	5.37	0.63
Bdna	8.42	5.67	2.66	0.09
Arc2d	31.09	21.84	3.77	5.48
Jpeg Decode	0.69	-	0.05	0.64
Epic	6.19	0.01	0.05	6.13
Jpeg Encode	4.56	-	0.04	4.52
Gsm Encode	3.95	-	0.01	3.94

Table 6.1 Percentage of total execution time that the vector cache is stalled and stall reason.

We can see in this table that, in general, numerical programs are affected by bottlenecks related to their large working sets. They make a higher use of the main memory bus,

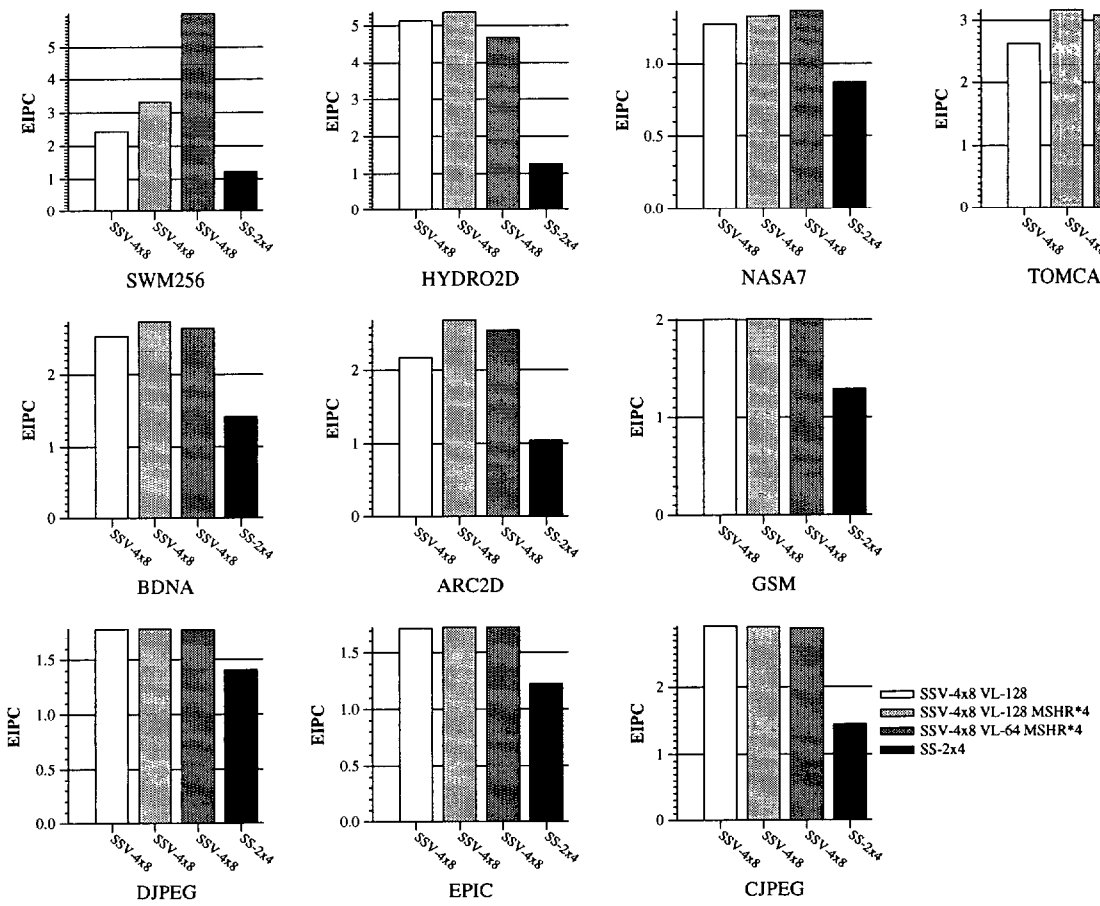


Figure 6.1 Overall performance results for the basic SSV-4x8, the SSV-4x8 enhanced with quadrupled MSHR and WB entries (both vector length 128 and 64 elements), and the SS-2x4 superscalar processor.

causing stalls in the vector cache when the MSHR or the WB entries fill up. Clearly, the programs that exert most pressure onto the RDRAM array are the ones that stall the most due to MSHR entries being full.

Therefore, in order to reduce this performance bottleneck, we have set a larger number of entries in the memory non-blocking mechanisms. This section evaluates performance after this improvement.

General performance

Figure 6.1 shows the performance obtained when the number of MSHR and Write Buffer entries is quadrupled. We observe in that figure that, in general, all numerical programs increase their performance as the memory non-blocking mechanisms are improved. The larger performance increases appear in *Swim256*, *Tomcatv* and *Arc2d*, which, as could be seen in table 6.1, are the programs that spend a larger fraction of their execution time stalled because of the vector cache. Moreover, by adding columns 3 and 4 of this table we can see that these programs are the most affected by the filling up of the MSHR and WB structures. Although in *Nasa7* the vector cache also spends 19.53% of the total execution time stalled due to the MSHR and the WB structures being full, its performance improvement when the number of entries of the MSHR and WB structures is enlarged is not that good. The reason is the wide use of non stride-1 vector memory accesses in this program, as each element of a non stride-1 vector memory load needs one MSHR entry (and one WB entry if it is a vector memory store).

Table 6.2 shows the total stall time and the reason of the stall for the execution of the ten benchmark programs in the upgraded SSV architecture. Note that the percentage values are referred to the lower execution times that are obtained in the SSV architecture with improved memory non-blocking mechanisms. As expected from observations in figure 6.1, the total stall time percentage has decreased in almost all programs. Comparing table 6.2 with table 6.1 we can evaluate the decrease in the stall time. We observe that now the filling up of the WB is not an important stall reason in any program. Only in two programs (*Swim256* and *Nasa7*) the filling up of the MSHR is still an important stall reason, halting the vector cache around 15% of the total execution time.

Performance inside D-regions

Figure 6.2 shows the performance inside D-regions for the basic SSV-4x8 architecture and the SSV-4x8 architecture enhanced with quadrupled MSHR and WB entries. The figure shows results for both 128-element and 64-element vector registers. It also shows the performance inside D-regions for the SS-2x4 architecture. In this figure we observe

	Total	Breakdown by stall reason		
		MSHR Full	WB Full	Coher.
Swim256	15.12	14.71	0.40	0.02
Hydro2d	1.19	-	0.23	0.96
Nasa7	16.61	15.14	1.13	0.34
Tomcatv	5.20	4.45	-	0.75
Bdna	2.01	1.49	0.40	0.12
Arc2d	10.39	2.42	0.58	7.39
Jpeg Decode	0.64	-	-	0.64
Epic	6.27	-	0.04	6.23
Jpeg Encode	4.97	-	-	4.97
Gsm Encode	3.94	-	-	3.94

Table 6.2 Percentage of total execution time that the vector cache is stalled and stall reason after increasing the MSHR and WB entries.

two different behaviors. On one hand, multimedia programs are not affected by the increase in the number of entries in the MSHR and WB structures. As already discussed, these programs are not limited by a short number of entries in these structures, so increasing them does not change performance. On the other hand, numerical programs increase their performance as the memory non-blocking mechanisms are improved. The performance of these programs is limited by the MSHR and WB number of entries, so that increasing them improves performance up to 37% for *Swim256* with 128-element vector registers, and up to 46% for the same program with 64-element vector registers.

In general, the increase is larger with 128-element than 64-element vector registers. In this case moving towards a shorter vector register implies a performance loss due to the increasing number of instructions and operations that must be executed. In *Hydro2d* this performance loss makes EIPC smaller than for basic SSV-4x8 architecture. For this program, increasing the memory non-blocking mechanism together with moving to a shorter vector length is not a good solution. It is better to keep the same MSHR and WB number of entries, with 128-element vector registers. Contrary to the rest of programs, *Swim256* and *Nasa7* improve their performance as the vector length is decreased. A shorter vector length implies a different way of accessing data and a shorter number of collisions in caches, so that this improvement, together with the increase in the MSHR and WB entries, overcomes the increase in the number of instructions and operations that must be executed.

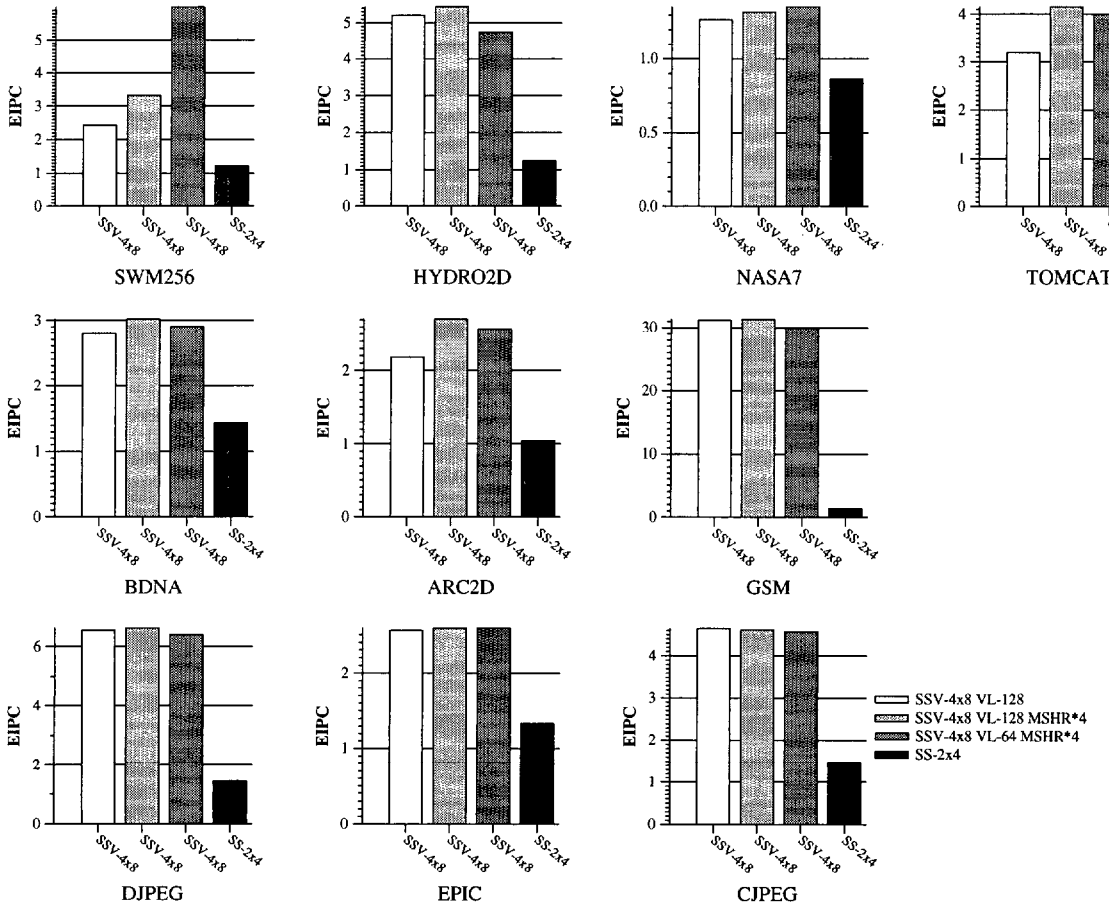
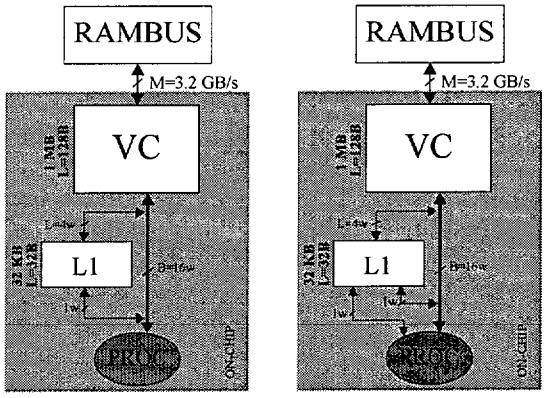


Figure 6.2 Performance results inside D-regions for the basic SSV-4x8, the SSV-4x8 enhanced with quadrupled MSHR and WB entries (both vector length 128 and 64 elements), and the SS-2x4 superscalar processor.

Summing up, increasing the memory non-blocking mechanisms in the SSV architecture is a good solution in order to diminish the negative effect of the vector cache stall due to the filling up of the MSHR and WB structures. This improvement will be added to the basic SSV architecture, so that the following configuration enhancements will be made starting from this non-blocking enhanced SSV architecture.



(CB) (CB_2p)

Figure 6.3 The CB and CB_2p memory models. CB_2p memory model is an enhanced CB model with an additional scalar memory port.

6.3 ADDITIONAL MEMORY PORT FOR SCALAR ACCESSES

As previously shown in table 3.13, page 83, there is a fair amount of scalar code in pure S-regions, especially for multimedia programs, and there is also a fair amount of scalar code within D-regions. Comparing the SSV-4x8 and the SS-2x4 architecture, a key limiting resource in the SSV architecture is probably the number of scalar memory references it can perform in parallel (1 scalar reference per cycle at most). Meanwhile, the SS-2x4 architecture can perform two independent memory references per cycle. In those pieces of vector code that include an important amount of scalar code it would be helpful to have an extra memory port devoted to the scalar accesses. Figure 6.3 presents the basic CB memory model against the CB memory model enhanced with an additional scalar port connected to the L1 data cache (named CB_2p memory model). In the CB_2p memory model we have one shared memory port for accessing both scalar and vector data, and one additional memory port for exclusively accessing scalar data. In the following sections we will evaluate this enhanced memory model from the performance point of view.

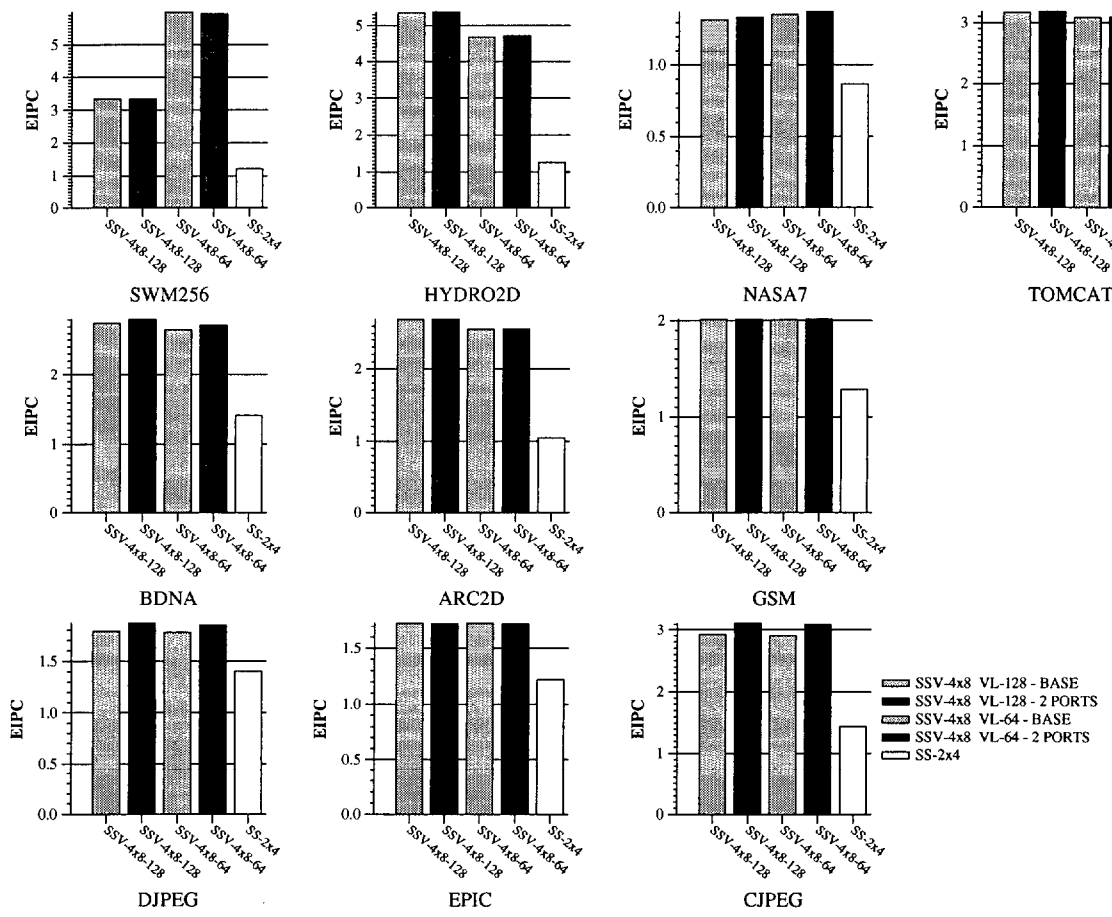


Figure 6.4 Overall performance results for the basic SSV-4x8, the SSV-4x8 enhanced with an additional scalar port and the SS-2x4 superscalar processor.

General Performance

In this section we evaluate the overall performance of the SSV architecture enhanced with an additional scalar memory port connected to the L1 data cache. Figure 6.4 shows the performance obtained for the previously non-blocking enhanced SSV architecture and for the SSV architecture with the CB_2p memory hierarchy attached. In both cases this figure shows results for the 128-element and 64-element vector registers. The figure also shows the performance obtained by the SS-2x4 architecture, which has not been enhanced with additional ports as this machine has already two independent memory ports.

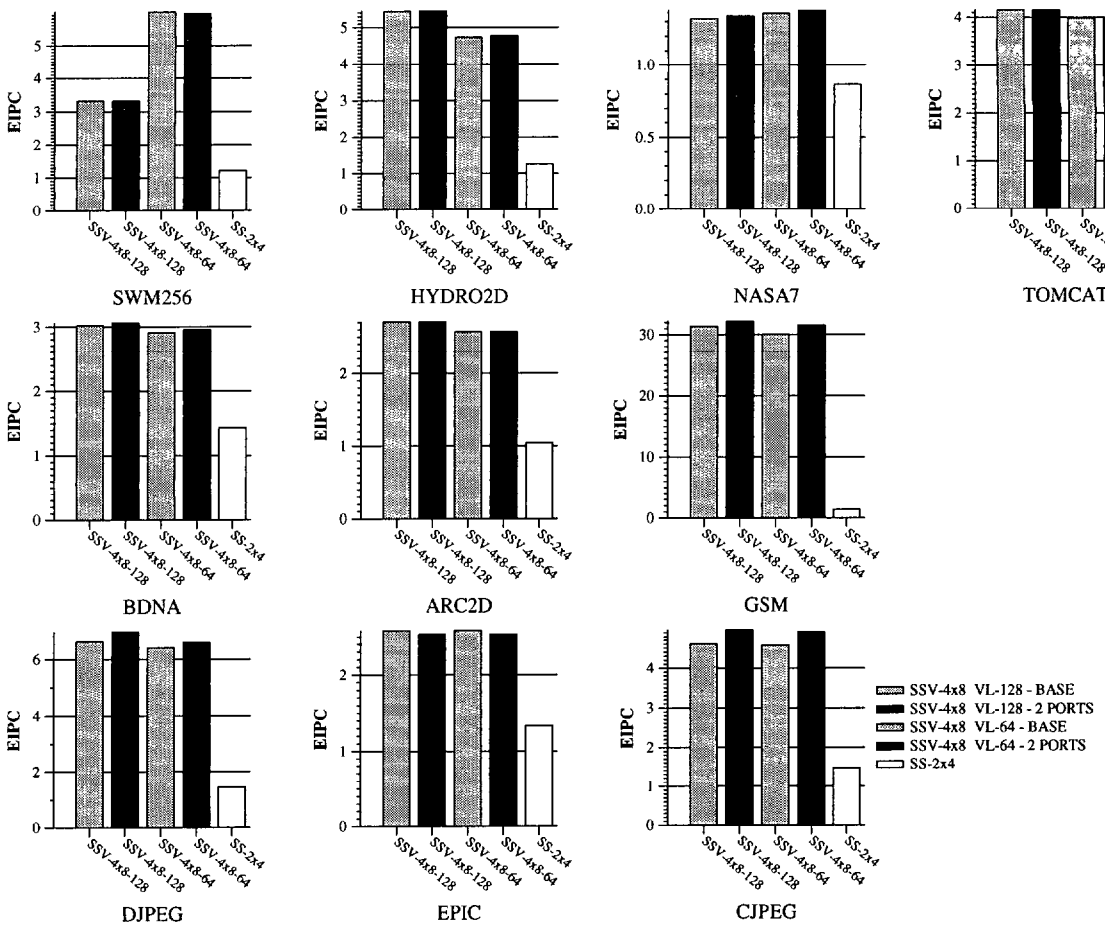


Figure 6.5 Performance results inside D-regions for the basic SSV-4x8, the SSV-4x8 enhanced with an additional scalar port and the SS-2x4 superscalar processor.

In figure 6.4 we observe that the larger performance improvements appear in multimedia programs, concretely in *Jpeg Decode* and *Jpeg Encode*, with improvements of 5% and 7%, respectively. The reason is that these programs have a large fraction of scalar code, so they can take advantage of having an extra scalar memory port in order to increase parallelism in data accesses. The extra port will be used inside both D-regions, in parallel with the vector data port, and inside S-regions. As discussed in section 3.9, page 82, the effect on overall performance will also depend on the relative importance of scalar code in the whole program.

Performance inside D-regions

Figure 6.5 presents the performance inside data-parallel regions when we add the scalar port just discussed. We can see in that figure that numerical programs hardly improve their performance. Slight improvements can be observed in *Nasa7* and *Bdna*. However, multimedia programs like *Jpeg Decode*, *Jpeg Encode* and *Gsm Encode* improve their performance inside D-regions in 6%, 8% and 5% respectively, which was expected since their vector regions are polluted with scalar instructions. The extra scalar port provides additional flexibility when accessing the L1 data cache for these programs. Although we could expect the same behavior for *Epic*, the coherency problems already discussed offset any gains that the extra cache port might offer.

In general, adding an extra port for scalar accesses is a profitable enhancement for running a heterogeneous set of programs in the ILP+DLP architecture, as it provides some additional flexibility in accessing memory. This extra port improves the execution of low vectorizable programs with D-regions containing a large amount of scalar instructions. We will include this enhancement in the SSV-4x8 architecture that will be used in the following sections as a basis for the comparison with the following tunings.

6.4 IMPROVING MAIN MEMORY BANDWIDTH

Another bottleneck that we identified in the previous chapter was the large amount of traffic that some programs generated to/from the RDRAM array. As we discussed in section 5.4.3, page 159, multimedia programs have low main memory traffic. However, numerical programs exert a higher pressure on the main memory, and their memory traffic reaches higher values. This traffic will be served at the sustained bandwidth that the main memory delivers. The effects of this bottleneck can be tackled in two different ways: first, increasing raw bandwidth; second, making sure that the available bandwidth is fully used by providing enough Miss Status Holding Registers and Write Buffers. To see the effect of these bottlenecks we have doubled the RDRAM bandwidth for both the SSV and the SS architectures, so the RDRAM main memory model now

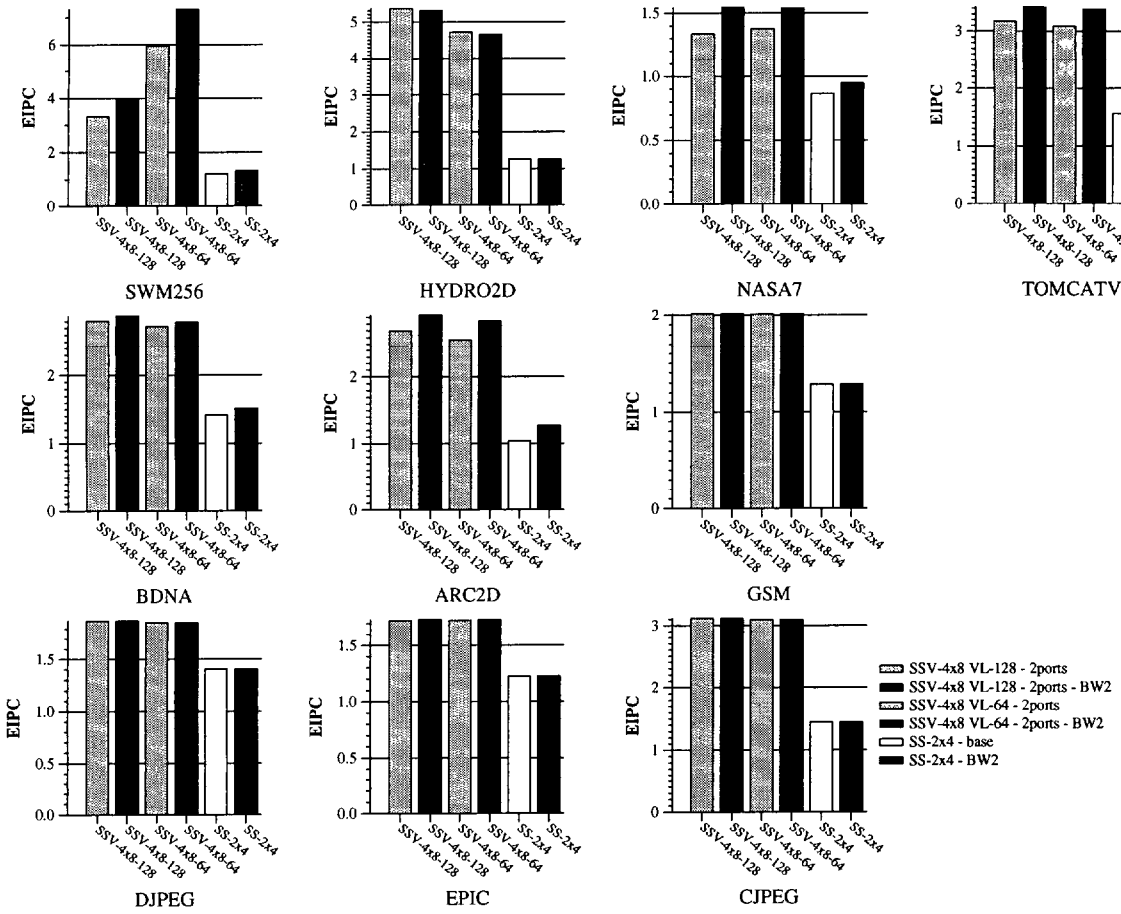


Figure 6.6 Overall performance results for the SSV-4x8 and one additional port, the SSV-4x8 enhanced with one additional scalar port and double memory bandwidth, MSHR and WB entries, the basic SS-2x4 superscalar processor and the SS-2x4 processor enhanced with double memory bandwidth, MSHR and WB entries.

delivers 6.4 GB/s. We have also doubled again the number of MSHR and Write Buffer entries in order to diminish the negative effect of filling these structures.

The following subsections study the effect in performance of improving bandwidth in the SSV and SS architectures. We will present overall performance and performance inside D-regions.

General Performance

Figure 6.6 shows the overall performance results for the basic SSV-4x8 architecture and the SSV-4x8 with double bandwidth, MSHR and WB entries. These values are presented for 128-element and 64-element vector registers. This figure also shows the performance values obtained for the basic SS-2x4 processor and the SS-2x4 processor enhanced with double bandwidth, MSHR and WB entries.

We can see in that figure that, as expected, multimedia programs are hardly affected by the increase in the memory bandwidth. Their small memory traffic was served at a high rate in the basic SSV configuration so that they do not benefit from the improved bandwidth.

Numerical programs, however, improve their performance when the memory bandwidth is doubled. The most important improvements appear in programs *Swim256*, *Nasa7*, *Arc2d* and *Tomcatv*, being as large as 20.2%, 15.5%, 9% and 7.8%, respectively, for vector length 128, and as large as 23%, 12%, 11% and 10% for vector length 64. We observe from these data that the 64-element vector registers reach higher performance values than the 128-element vector registers for numerical programs, while for multimedia programs their performance is very similar. We also observe from these data that the higher bandwidth benefits those programs affected by either large working sets (*Swim256* and *Tomcatv*), or non stride-1 vector memory accesses (*Nasa7* and *Arc2d*).

After performing all these tunings the vector cache stall time has been considerably reduced, as can be observed in table 6.3. We can see in this table that the only reason that stalls the vector cache for an important percentage of the total execution time is the coherence preservation between the L1 and L2 data cache. The other stall reasons have been reduced down to 1% of the total execution time.

In figure 6.6 we also observe that, for the superscalar architecture, the only programs affected by doubling the main memory bandwidth are *Swim256*, *Nasa7*, *Bdna* and *Arc2d*, with improvements of 9%, 9.5%, 7% and 21%, respectively. All in all, the

	Total	Breakdown by stall reason		
		MSHR Full	WB Full	Coher.
Swim256	0.94	0.83	0.08	0.02
Hydro2d	1.00	-	0.05	0.95
Nasa7	1.88	0.85	0.32	0.70
Tomcatv	1.27	0.47	-	0.80
Bdna	0.16	0.01	-	0.15
Arc2d	7.27	0.06	0.06	7.14
Jpeg Decode	0.68	-	-	0.68
Epic	6.43	-	-	6.43
Jpeg Encode	5.21	-	-	5.21
Gsm Encode	4.02	-	-	4.02

Table 6.3 Percentage of total execution time that the vector cache is stalled and stall reason, after increasing non-blockingness, adding one scalar memory port and doubling the main memory bandwidth (and non-blockingness, again).

performance values that the SS architecture achieves are very far from those reached by the SSV architecture.

Performance inside D-regions

Figure 6.7 presents the performance inside D-regions for the basic SSV-4x8 architecture and the SSV-4x8 enhanced with double bandwidth, MSHR and WB entries, for 128-element and 64-element vector registers. It also shows performance inside D-regions for the basic SS-2x4 architecture and the SS-2x4 architecture with doubled bandwidth, MSHR and WB entries.

This figure is very similar to the previous one since multimedia programs are hardly affected by doubling the main memory bandwidth and the high vectorization percentage of numerical programs makes overall results very similar to those inside D-regions.

Numerical programs take better advantage of doubling main memory bandwidth with improvements comparable to those commented for the overall performance. As we increase the RDRAM bandwidth in the SSV-4 architecture, programs limited by memory bandwidth, such as *Swim256*, *Nasa7*, *Arc2d* and *Tomcatv* increase their performance in

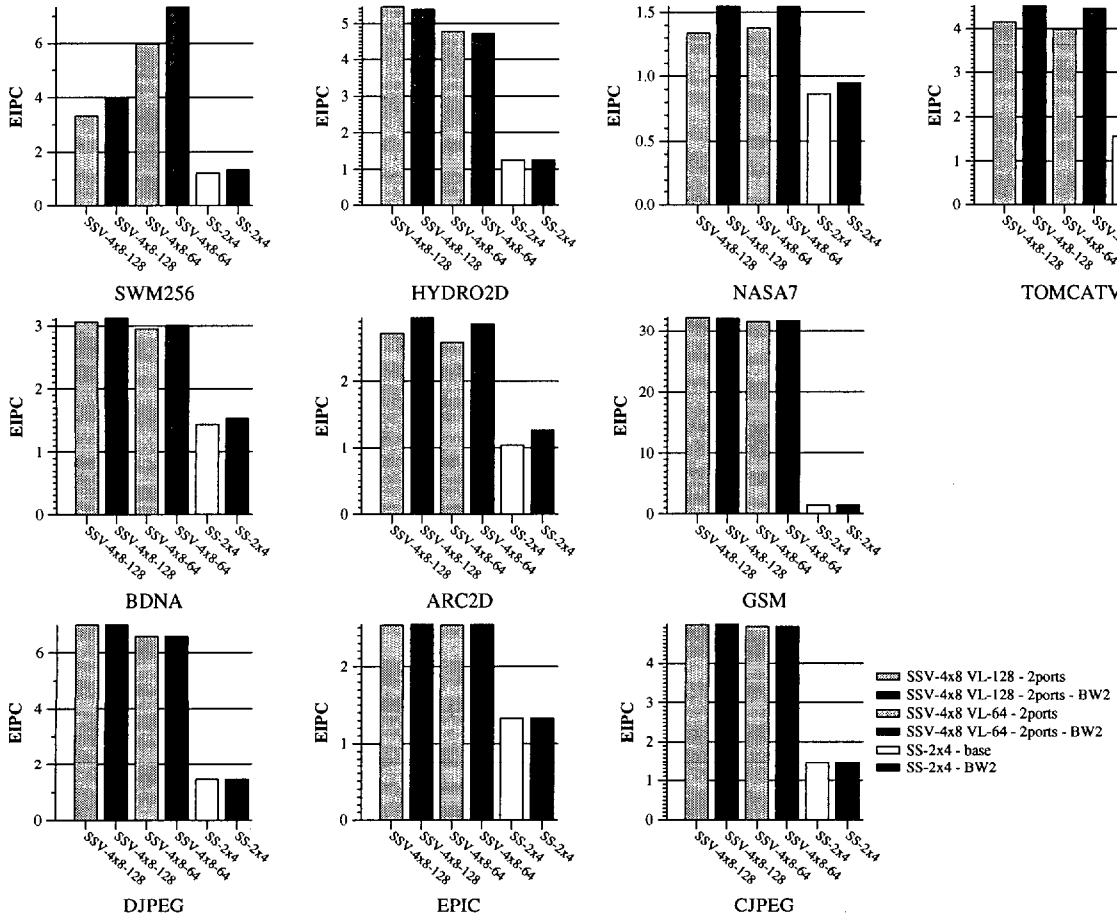


Figure 6.7 Performance results inside D-regions for the SSV-4x8 with one additional port, the SSV-4x8 enhanced with one additional scalar port and double memory bandwidth, MSHR and WB entries, the basic SS-2x4 superscalar processor and the SS-2x4 processor enhanced with double memory bandwidth, MSHR and WB entries.

20.3%, 15.4%, 9% and 8.6%, respectively. The rest of the programs, as expected, are not affected.

The SS-2x4 architecture presents similar improvements for the same set of programs. Although in some of these programs the relative improvement could be greater than in the SSV architecture, the SS-2x4 IPC values are still very far from those of SSV-4x8 EIPC.

	1MB CB (base)	2MB CB	4 MB CB
Size	1MB	2MB	4 MB
# Sets	4096	4096	4096
Line Size	128B	128B	128B
Associativity	2	4	8
Latency (cycles)	4	8	12
MSHR entries	64	64	64
WB depth/retire	64/32	64/32	64/32
Mem Bandwidth	6.4 GB/s	12.8 GB/s	12.8 GB/s

Table 6.4 Cache Hierarchy Parameters for the basic and the two enhanced vector caches: *WB depth/retire* is the number of Write Buffer entries/retire-at-X policy.

Summing up, doubling the main memory bandwidth is a way to overcome the performance bottleneck found in programs with a high pressure on the main memory, because of their large working sets or because of non stride-1 vector memory accesses.

6.5 EFFECTS OF MICROPROCESSOR INTEGRATION

Although current-generation superscalar processors typically have a large on-chip L2 cache (such as the Alpha 21364 [Ban98] that has a 1.75 MB L2 cache), advances in logic integration are allowing more and more transistors to be integrated on a chip. Therefore, a larger die area can be devoted to the L2 cache in future-generation processors. That is the case, for example, of the Alpha 21464 processor [Eme99] that includes 250 million transistors inside and will implement an even larger L2 data cache. These larger L2 caches will be implemented by using more memory banks, which will favor a natural increment in the cache associativity, and, unfortunately, also in the cache latency.

Future improvements will also provide with a higher memory bandwidth. Following the example of Alpha 21464 [Eme99], its 1100 signal pins will allow the memory to transfer data to the processor at, at least, 12 GB/s. IBM Power4 [Die99], with its up to 32 MB off-chip L3 cache, and a 16-byte width L3 port also provides over 10 GB/s of memory bandwidth.

This section studies the performance behavior of the SSV architecture as the size, associativity and latency of the L2 cache is increased. We will use the CB memory model presented in table 5.3 as the basic memory configuration, enhanced with double bandwidth (6.4 GB/s), 64 Miss Status Holding Registers, 64 Write Buffer entries and one extra scalar memory port. We will improve this memory configuration by doubling the size of the L2 cache and increasing the cache associativity and latency. The L1 data cache will not be changed, and it will keep the configuration already shown in table 5.3. We will also increase the main memory bandwidth to 12.8 GB/s. The main configuration parameters of both enhanced memory designs are presented in table 6.4. We can see in this table the basic configuration and the new 2MB and 4MB cache configurations. As the L2 cache size is increased more memory banks are added to the cache, so associativity is increased without additional effort. Unfortunately, this increase also causes an increase in the L2 cache latency.

General Performance

Overall performance results, presented in figure 6.8, show that the effect of increasing the L2 cache parameters (size, associativity and latency), as well as the main memory bandwidth, benefit especially those programs limited by memory. *Swim256*, *Nasa7*, *Tomcatv*, *Bdna* and *Arc2d* improve their performance in 13%, 13%, 15%, 3% and 5%, respectively, for the 2MB vector cache and 128-element vector registers. The use of 64-element vector registers also improves performance for the same programs, although in slightly lower percentages (from 2.2% to 10.5%). The reason of this behavior is that the larger L2 cache latency does not harm performance as the increase in the L2 cache size and associativity, as well as the higher memory bandwidth, make up for the extra cycles that the processor has to pay in every L2 cache access.

However, those programs which are not limited by memory do not compensate the larger number of cycles that every L2 cache access costs with the decrease in the number of main memory accesses, so that, the overall performance diminishes. All in all, the performance loss is low, being as large as 1.8%, 3.2%, 3% and 0.1% for *Hydro2d*, *Jpeg Encode*, *Epic* and *Jpeg Decode*, regardless of the vector length used. *Gsm Encode*

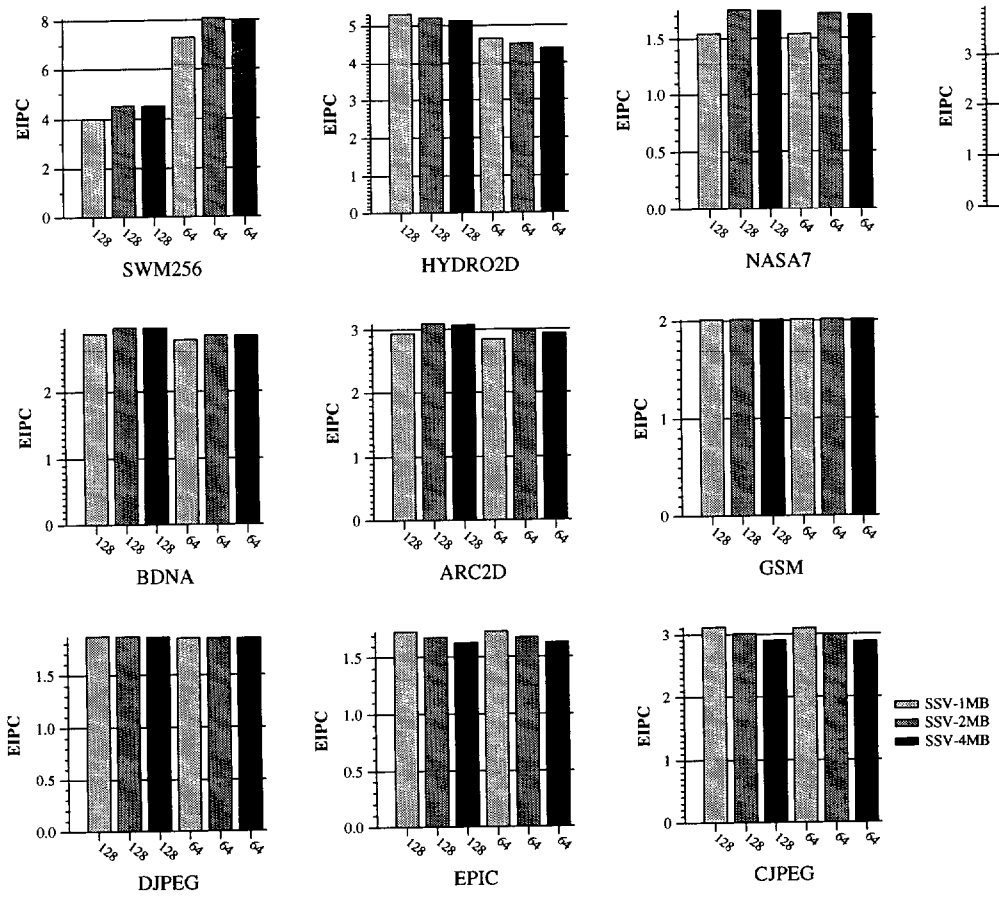


Figure 6.8 Overall performance results for the sophisticated SSV-4x8 backed with 1MB, 2MB and 4MB vector caches.

does not modify its performance neither with 128-element nor with 64-element vector registers.

When the L2 cache size and associativity are increased again to 4MB and 8-way the behavior repeats. Programs limited by memory, that is *Swim256*, *Nasa7*, *Tomcatv*, *Bdna* and *Arc2d* increase their performance in 12.4%, 12.5%, 11.1%, 2.9% and 4.5%, respectively, for vector length 128, over the base configuration. Increases for vector length 64 go from 2.1% up to 10.7%. *Gsm Encode* undergoes a slight slowdown of 0.04% for vector length 128 and 64, over the base configuration, and the rest of the programs undergo an even larger performance decrease (from 0.4% for *Jpeg Decode* up to 7% for *Jpeg Encode*).

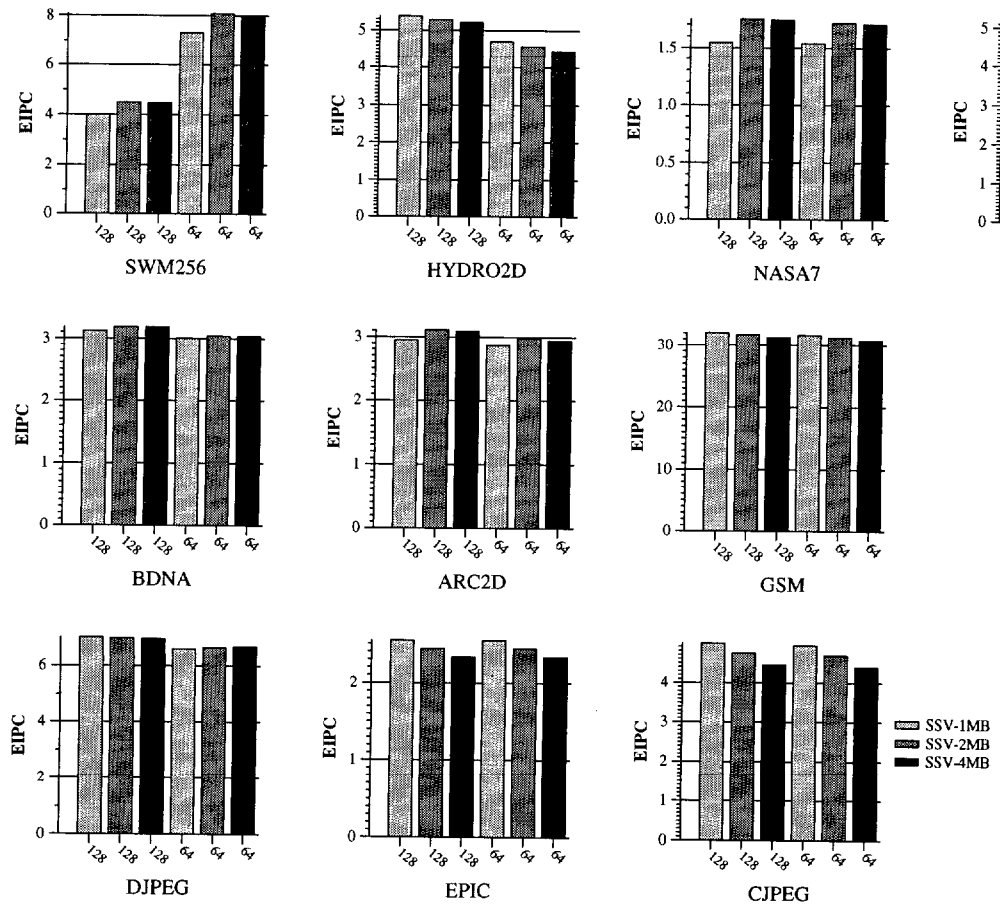


Figure 6.9 Performance results inside D-regions for the sophisticated SSV-4x8 backed with 1MB, 2MB and 4MB vector caches.

Comparing the 2MB and 4MB results we realize that the 2MB performance values are only a little bit larger than the 4MB results. Although the cache latency has undergone a 50% increase (from 8 cycles to 12 cycles) the performance results are hardly affected. Therefore, we have reached a configuration point such that, with that higher latency, and with the same processor cycle time, it does not pay off spending more chip area in increasing the cache size.

Performance inside D-regions

The behavior inside D-regions is similar to the overall performance, taking into account that measures inside data-parallel regions reach higher EIPC values as they do not include the effect of the S-regions.

Similarly to the overall performance, figure 6.9 shows that programs that are constrained by the memory hierarchy improve their performance as the L2 cache parameters are improved. Meanwhile, the rest of the programs suffer a performance loss because of the influence of the higher latency over the reduced main memory traffic.

Considering the same processor frequency, the 4MB vector cache presents a slight performance slowdown over the 2MB vector cache. However, if the processor cycle time is decreased the slowdown could turn into a performance increase.

6.6 ATTACKING THE STRIDE PROBLEM: COLLAPSING AND MULTI-ADDRESS SECONDARY CACHES

The previous sections have grappled with the extra memory traffic that the SSV architecture undergoes by using a rough approach: double RAMBUS bandwidth, double MSHR and WB entries, and larger caches. However, as we have also seen section 3.7.3, page 71, a good portion of that traffic is due to non stride-1 memory accesses, which our vector cache, designed for simplicity, does not handle very well. In this section we evaluate two alternative cache hierarchies aimed at improving the performance of non stride-1 vector memory accesses.

The question is, do we have enough die area to implement more complex cache designs? In the previous section we have studied the evolution of performance as technology evolves and both, the size of the L2 data cache and the main memory bandwidth are increased. However, as technology evolves, this extra on-chip space can be used to

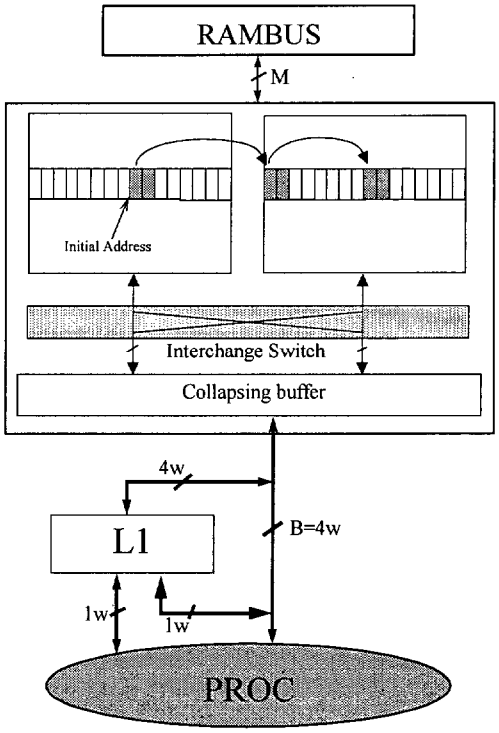


Figure 6.10 The Collapsing Vector Cache (CVC) data path.

implement more complex cache designs, rather than simply increasing the cache size and the memory bandwidth.

The first alternative design, shown in figure 6.10, is the Collapsing Vector Cache (CVC) [CEV99]. It uses a collapsing buffer (proposed by Conte et. al. [CMMP95]), that is able to retrieve several vector elements along two consecutive cache lines, even if they are not consecutively allocated. Instead of *shift and mask* logic, the collapsing buffer logic groups the requested elements together. This design will be useful for vector strides between 2 and 2×cache line size-1, as in these cases there will be more than one element of the vector access in the two consecutive cache lines that can be easily retrieved and delivered to the processor. For vector strides equal or larger than twice the cache line size the collapsing vector cache does not add any improvement to the basic vector cache already evaluated in previous sections.

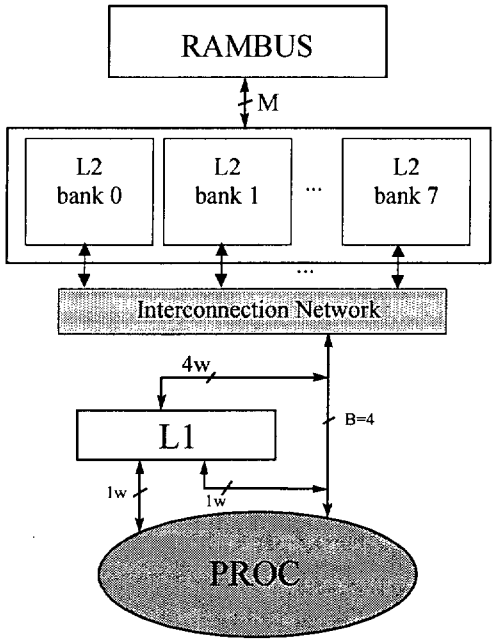


Figure 6.11 The Multi-Address Cache (MAC) data path.

	VC	CVC	MAC
# ports	1x4w	1x4w	4
# banks	2	2	8
Latency	4	6	8

Table 6.5 Configuration parameters for the basic Vector Cache (VC), the Collapsing Vector Cache (CVC) and the Multi-Address Cache (MAC).

The second alternative, shown in figure 6.11, is the Multi-Address Cache (MAC) [CEV99]. It is a conventional cache where a vector memory access is decoupled among all available memory ports. This model fully takes advantage of all the port resources, and we can send independent memory addresses no matter the stride between them. Therefore, this alternative is more flexible than the Collapsing Vector Cache as the access to the different elements of the vector does not depend on the value of the vector stride. Of course this design is even more complex and expensive than the Collapsing Vector Cache, but it will show the potential of the performance achievements that can be obtained with a suitable memory hierarchy.

Table 6.5 shows the configuration parameters for the three cache designs, in terms of number of ports, banks and latency. Note that we increase latency to the L2 cache as a way to account for the extra complexity of these cache designs. Remaining parameters do not change, that is, we have a 6.4 GB/s main memory bandwidth, 64 Miss Status Holding Registers, 64 Write Buffers (retire-at-32 policy) and the extra scalar memory port.

6.6.1 General Performance

Figure 6.12 presents the overall performance results for the enhanced SSV-4x8 architecture backed with the basic vector cache, the collapsing vector cache and the multi-address cache. The figure shows results for 128-element and 64-element vector registers. We observe that both, the CVC and MAC improve performance. A more detailed analysis of the figure reveals that:

- The Collapsing Vector Cache improves performance from 5% up to 90.6% for 128-element vector registers. The larger performance improvements appear in multimedia programs *Jpeg Decode*, *Gsm Encode*, *Jpeg Encode* and *Epic*, with values of 90.6%, 75.7%, 46.1% and 21%. Numerical programs achieve smaller improvements, being as large as 18.2%, 17.5%, 10.4% and 9.2% for *Swim256*, *Nasa7*, *Arc2d* and *Tomcatv*. *Bdna* reaches 5% improvement. Meanwhile, 64-element vector registers achieve slightly lower improvements in the range (4.2%, 90.4%).

  The exception to this general behavior is program *Hydro2d* as its performance decreases when the sophisticated Collapsing Vector Cache is attached to the SSV processor. This program does not benefit enough from accessing memory with vector strides in the (2, 2×cache line size-1) range, so that the increasing in the L2 cache latency causes a final performance decrease.

- The Multi-Address Cache reaches larger performance improvements than the Collapsing Vector Cache, reaching values in the (16.4%, 128%) range for the 128-element vector registers. Programs *Arc2d*, *Swim256*, *Jpeg Decode*, *Gsm Encode* and *Nasa7* achieve the largest EIPC values, with improvements of 128%, 127.5%, 90.7%, 79.8% and 67.6% respectively over the basic vector cache configuration. These pro-

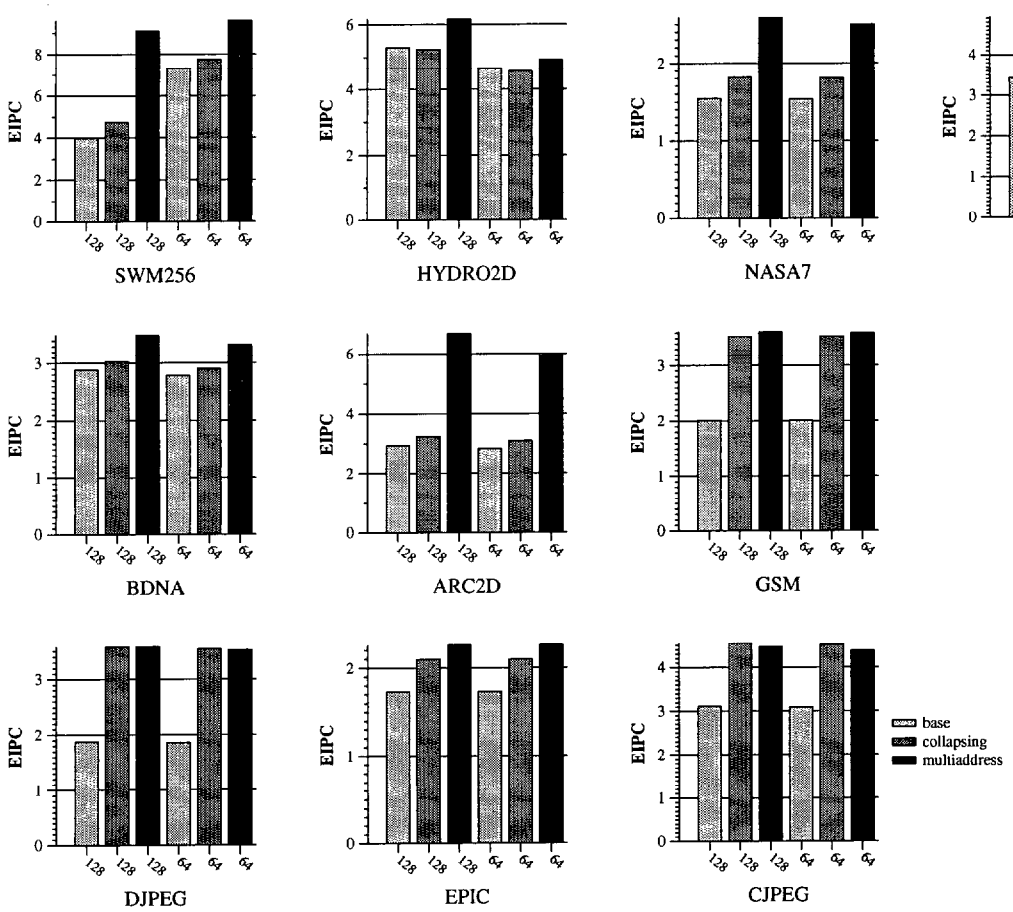


Figure 6.12 Global performance results for the enhanced SSV-4x8 architecture backed with the base vector cache (the 1MB vector cache from the previous section), a collapsing vector cache, and a multi-address cache.

grams exploit the benefit of the Multi-Address Cache that is not restricted to a certain value range in the vector stride, but favors all the vector strides.

Although they are not as large as these values, important improvements are also achieved in programs *Tomcatv*, *Jpeg Encode*, *Epic*, *Bdna* and *Hydro2d*, with values 44.5%, 43.4%, 30.4%, 21.3% and 16.4% respectively. Among these programs, *Jpeg Encode* has a special behavior, as its improvement is larger in CVC than in MAC. The fact is that in this program the use of vector strides equal or larger than twice the cache line size is not large enough to be worthwhile including a more sophisticated cache design with a larger cache latency. Therefore, the larger latency

is the constraining point that makes the MAC design obtain slightly lower EIPC values.

For 64-element vector registers the performance improvements are slightly lower, still achieving values in the (5.8%,109%) range.

In general, in the MAC design all programs improve their performance, meaning that in spite of the larger memory latency, programs achieve a decrease in the overall memory access time due to the easibility of strided vector memory accesses.

The analysis of figure 6.12 has shown that both the CVC and MAC designs allow improving the performance in those programs that use strided vector memory accesses or are limited by memory. MAC obtains better performance results, even though it has a larger cache latency, due to its more flexible design.

6.6.2 Performance inside D-regions

Figure 6.13 presents the performance results inside D-regions for the three cache hierarchies described above. As we provide more flexibility in the number and types of memory accesses we allow, performance improves accordingly. Comparing this figure with the previous one, we can see that although the general trends are very similar, programs *Tomcatv*, *Gsm Encode*, *Jpeg Decode*, *Epic* and *Jpeg Encode* reach higher EIPC values that are then soften in the overall performance when the performance inside S-regions is also included.

The Collapsing Vector Cache improves performance in almost all programs, from 0.5% to 45.8% for both 128-element and 64-element vector lengths. Improvements are concentrated in programs that have strides 2, 3 and 4, which benefit from the collapsing hardware.

The Muti-Address Cache improves performance for all programs. The larger improvements happen in programs that either have very large strides (*Bdna* 23.6%, *Nasa7* 67.6% and *Arc2d* 129.5%), or are strongly memory limited (*Swim256* 128% and *Tomcatv* 76.6%).

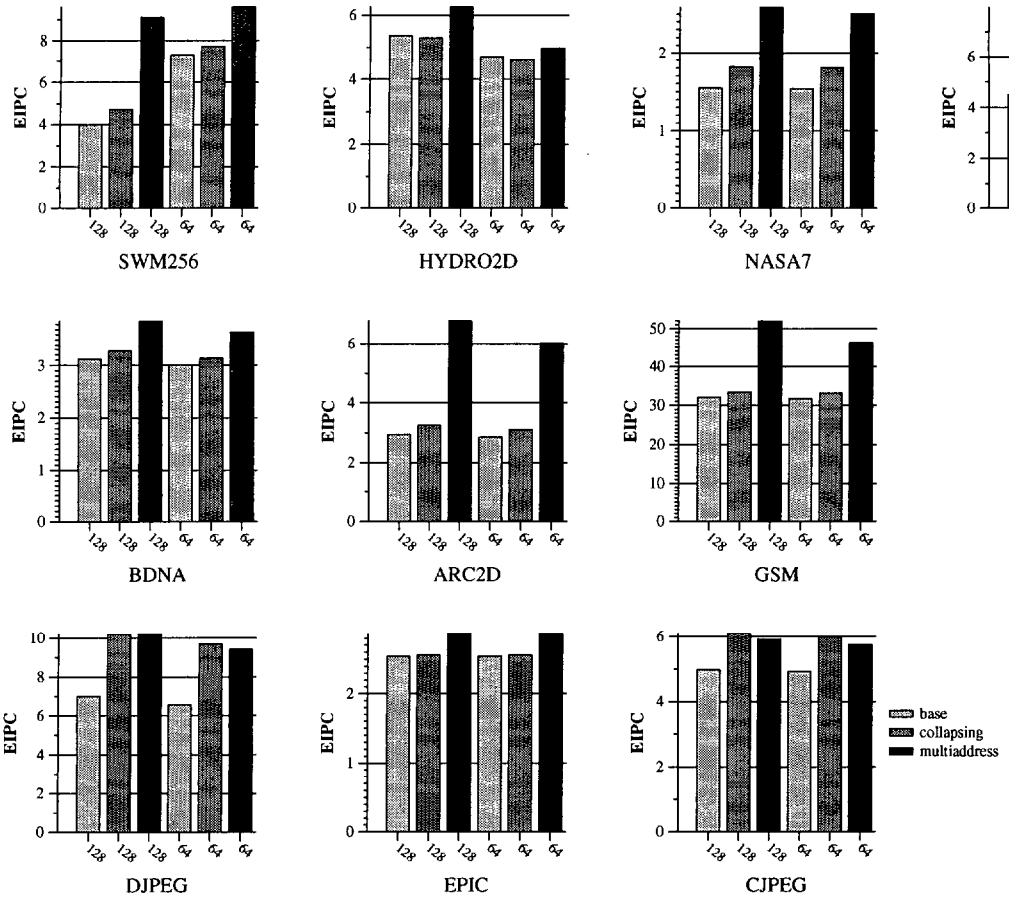


Figure 6.13 Performance results inside D-regions for the enhanced SSV-4x8 architecture backed with the base vector cache (the 1MB vector cache from the previous section), a collapsing vector cache, and a multi-address cache.

All in all, we have seen in this section that large performance improvements can be obtained as the cache design is tuned in order to tackle strided memory accesses. Although these cache designs are complex and expensive to implement, the large performance improvements obtained pay off the effort, so that as technology evolves they can actually be implemented.

6.7 SUMMARY

The studies in the previous chapter evidenced some bottlenecks related to the memory hierarchy design. These bottlenecks mainly appeared in numerical programs with either large working sets or strided memory accesses. These programs had a large amount of traffic with the main memory, a relatively low hit rate, and a vector cache stall time that in some cases reached up to 60% of the total execution time.

In this chapter we have studied how to decrease the effect of these bottlenecks and improve the performance of the SSV architecture by tuning the memory hierarchy. For this study we first selected the CB memory model because of its advantages when compared to the CA memory model, and then we have analyzed different ways of diminishing the negative effects of the performance bottlenecks examined in the previous chapter.

The first enhancement to the baseline SSV architecture plus CB memory model under study has been the additional number of Miss Status Holding Register and Write Buffer entries. Increasing the memory non-blocking mechanisms in the SSV architecture diminishes the negative effect of the vector cache stall due to the filling up of the MSHR and WB structures. The performance improvements obtained with this tuning reaches up to 37% for 128-element vector registers (46% for 64-element vector registers).

We also studied the possibility of adding an extra scalar memory port to the SSV architecture. We realized that probably, a key limiting resource in the SSV architecture is the number of scalar memory references it can perform in parallel (1 scalar reference per cycle at most). Therefore, in those pieces of vector code that include an important amount of scalar code it would be helpful to have an extra memory port devoted to the scalar accesses.

Results show that adding an extra port for scalar accesses is a profitable enhancement for running a heterogeneous set of programs, which could even include low vectorization programs, in the SSV, as it provides with some additional flexibility in accessing memory.

Another improvement that we have studied in this chapter is the increase of the memory bandwidth. It has been motivated by the large amount of traffic that some programs generated to/from the RDRAM array. Although multimedia programs have low main memory traffic, numerical programs exert a higher pressure on the main memory, and their memory traffic reaches higher values. For this reason numerical programs have taken the better advantage of doubling main memory bandwidth. As we increase the RDRAM bandwidth in the SSV-4 architecture, programs limited by memory bandwidth, such as *Swim256*, *Nasa7*, *Arc2d* and *Tomcatv* increase their performance up to 20%. The rest of the programs, as expected, are not affected.

We made the same memory bandwidth improvement in the SS architecture, and we observed that the SS-2x4 architecture presents similar improvements for the same set of programs. Although in some of these programs the relative improvement could be greater than in the SSV architecture, the SS-2x4 IPC values are still very far from the SSV-4x8 EIPC values.

Finally, we also studied in this chapter two possible future evolutions of the vector cache. The first one consists in increasing its size, associativity and latency, at the same time that the memory bandwidth is also increased. This analysis has shown that programs that are constrained by the memory hierarchy improve their performance as the L2 cache parameters are improved. Meanwhile, the rest of the programs undergo a performance loss because of the prevalence of the higher latency over the reduced main memory traffic.

When the cache latency is increased again up to 12 cycles our results demonstrate that, considering the same processor frequency, the 4MB vector cache presents a slight performance slowdown with respect to the 2MB vector cache. However, if the processor cycle time is considered to decrease, the slowdown could turn into a performance increase.

The second future enhancement of the vector cache consists in using the extra on-chip space in order to implement more complex cache designs, rather than simply increasing the cache size and the memory bandwidth. In particular, we evaluated two additional

cache designs [CEV99] aimed at better dealing with the non stride-1 memory references found in both numerical and multimedia programs. Simulations show that these new designs deliver improvements with respect to the base case that range from 5% up to 128%.

The Collapsing Vector Cache (CVC) uses a collapsing buffer (proposed by Conte et. al. [CMMP95]), that is able to retrieve several vector elements along two consecutive cache lines, even if they are not consecutively allocated.

The Multi-Address Cache (MAC) [CEV99] is a conventional cache where a vector memory access is decoupled among all available memory ports. As we can send independent memory addresses regardless of the stride between them, this alternative is more flexible and expensive to implement.

The study has shown that both the CVC and MAC designs, although with higher cost than the baseline vector cache, allow improving the performance in those programs that use strided vector memory accesses or are limited by memory. The Collapsing Vector Cache improves performance in almost all programs, from 5% up to 90% for both 128-element and 64-element vector lengths. Improvements are concentrated in programs that have strides 2, 3 and 4, which benefit from the collapsing hardware. MAC obtains better performance results, even though it has a larger cache latency, due to its more flexible design. The larger improvements happen in programs that either have very large strides (up to 127.5% in *Arc2d*), or are strongly memory limited (up to 128% in *Swim256*).

Therefore, large performance improvements can be obtained as the cache design is tuned in order to tackle the strided memory accesses. Although these cache designs are complex and expensive to implement, the large performance improvements obtained pay off the effort, so that as technology evolves they can actually be implemented.

7

CONCLUSIONS AND FUTURE WORK

Summary

This chapter summarizes the main contributions of this thesis. Future lines of work are also discussed.

7.1 CONCLUSIONS

We started this thesis realizing that the scalability of superscalar processors is expensive and strongly technology-dependent. Current superscalar architectures can not be improved by just scaling up the number of instructions that are fetched, decoded, issued and committed. Current trends in the exploitation of available parallelism support this statement, and the design of current and near-future processors is currently exploiting new ways of parallelism. These trends identify the incipient use of ILP coupled with other ways of exploiting parallelism, such as simultaneous multithreading or chip multiprocessing.

Based on this analysis we have presented the data-level parallelism paradigm as an alternative way to exploit a different style of parallelism. Vector instructions have several inherent advantages: a lower number of instructions and operation executed, reduced pressure in the instruction fetch unit, simplicity of the control unit, advance knowledge of memory accesses (which can be scheduled in a better way), use of 100% of the requested data, ability to amortize functional units and memory startup latencies and simplicity to be scaled up by replicating functional units.

All these advantages have lead us to consider the exploitation of DLP as a way to improve current ILP architectures. However, we identified first the reasons why the success of traditional DLP architectures, that is vector architectures, was declining some years ago. They can be summarized in the fact that they are not as general purpose and their price/performance ratio worsened when compared to the other paradigms of supercomputing. While absolute performance of vector machines increased, the memory systems required to keep these powerful machines fed with data were still large, complex and very expensive. Thus, when comparing vector mainframes against multiprocessors using CMOS cache-based superscalar microprocessors, the cost per megaflop clearly favors the CMOS multiprocessors. Moreover, cache based superscalar machines were perceived to be more flexible than vector architectures, since the latter only perform well applications that are highly vectorizable. While cache based machines can not successfully execute all programs with very large data sets, their success in capturing a large fraction of most application domains is unquestionable.

Nowadays, parallel vector machine vendors offer supercomputing to the user at a much lower cost and with a higher performance. They have moved from the expensive ECL technology to CMOS technology, which lowers the fabrication costs and the power consumption appreciably. They have also migrated from expensive SRAM to commodity SDRAM chips, which provide a higher memory density, lower cost and higher performance.

Besides, the DLP paradigm has been also recently introduced in commodity microprocessors. However, a restricted form of vector computing aimed at exploiting sub-word level parallelism has only been introduced. In this thesis we do not focus on sub-word level parallelism. Rather, we integrate a full vector unit in an out-of-order superscalar processor. The advantage is that numerical applications can benefit greatly from them while, typically, do not take advantage of MMX-like ISAs. Furthermore, some multimedia applications that are not amenable to exploit sub-word parallelism can also take advantage of our vector units.

The main contribution of this thesis is to show that ILP and DLP can be merged in a single architecture to execute numerical and multimedia applications at a performance level that can not be achieved using either paradigm on its own.

ISA analysis

Our analysis of the instruction set architecture level has shown that, as expected, vector programs execute fewer basic blocks, instructions and operations than scalar programs due to the higher semantic content of vector instructions. Moreover, the larger the vector length, the shorter the number of instructions and operations executed. However, large vector lengths need larger vector register files, which are chip area consuming .

Another novel contribution of this thesis is the identification, separation and study of the S- and D-regions of each program. D-regions contain those pieces of code that can be vectorized, and S-regions contain pieces of code that are not amenable to be expressed using vector instructions. The ability for identifying and separately studying the behavior of a program inside D-regions and S-regions has allowed us to predict and

understand its performance behavior. The study inside regions at the instruction set architecture level showed that superscalar programs execute many more basic blocks, instructions and operations inside D-regions than the vector programs. The analysis of the S-regions exposed the problem that the quality of the scalar code generated by the superscalar compiler is higher than the quality of the scalar code generated by the vector compiler.

The solution consists in building a new set of benchmark programs starting from the pure superscalar and vector programs. Each hybrid vector program consists in the original S-regions from the superscalar version plus the original D-regions from the vector version. The hybrid vector programs execute fewer instructions and operations than the pure vector programs while they keep almost the same vector characteristics.

These initial studies at the ISA level lead us to conclude that, given the benefits of the vector ISA, it is worthwhile exploring the possibility of including a vector functional unit in a current superscalar architecture.

Proposed Architecture

The design of the ILP+DLP architecture is very similar to a current superscalar processor. The main difference comes from the addition of a Vector Register File (VRF) and its connections to the functional units present in the architecture. We have also added some special purpose registers: the vector length, vector stride, vector first and vector mask registers

Another contribution of this thesis is the design on the memory hierarchy, which is based on a new cache design, called "vector cache", which is able to deliver small vectors to the processor through a wide path. The vector cache is a dual banked memory from which full lines are read. These lines are switched, shifted and masked as necessary, and then data are sent to the processor. In this way, a high bandwidth, low latency data path to the memory is achieved.

Among the different ways in which the vector cache can be included in the memory hierarchy, we have explored two options: either include it as the L1 cache, or include it as the L2 data cache adding a direct path from the vector cache to the processor.

Scalability Study

We have presented the performance results of our proposed superscalar architecture with a vector unit and we have compared it with a traditional superscalar processor. We have studied two the scalability and potential performance of the SSV and SS architectures when a perfect memory is used. We have observed that the SSV architecture scales very well as more memory and computing resources are added to the processor. Moreover, it reaches higher values of parallelism than the SS architecture with a lower cost and control complexity.

The analysis inside D-regions and S-regions has shown that the SSV architecture achieves high values of parallelism inside D-regions. However, its contribution to the overall performance is determined by the relative weight of D-regions in the whole programs. For highly vectorizable programs, the overall performance is mainly determined by the high performance values obtained inside D-regions. However, low vectorizable programs have an overall performance that is also determined by the performance inside S-regions. As was expected, the performance inside S-regions is better for the SS architecture, since the superscalar core of the SSV architecture remains constant along the different configurations, while, in the SS architecture, the core is scaled adding more and more resources.

The study of the amount of data parallelism that each architecture is able to reach inside D-regions has shown that, although both, the SSV and the SS architectures, obtain larger OPC values as the configurations are scaled, the SSV architecture achieves larger values than the SS architecture in all programs.

Real Memory System Study

We have studied the SSV when a real cache hierarchy is introduced. We studied a cache hierarchy, called CA, where where the vector is in the first cache level. In contrast, in the CB model, the vector cache is in the second cache level.

The traffic study showed that regardless of the memory model, CA or CB, the cache hierarchy is able to filter a substantial portion of the processor's memory traffic. Multimedia programs have a very low memory traffic, and numerical programs present a lower amount of memory traffic when the CB model is used. Meanwhile, the results of the hit/miss study showed that multimedia programs hit much more often in cache than numerical programs. Again, the CB model reaches higher hit rate percentages. As a conclusion, numerical programs make a higher pressure on the main memory and the CB model, with a larger vector cache, reacts much better to this pressure with lower main memory traffic and higher hit rates.

The study of the vector cache stall time has revealed that this cache can spend up to 60% of the total execution time stalled due to different reasons. The two main reasons are full MSHR and WB entries. The other two reasons (coherence and cache conflicts) contribute in a lower degree to that bottleneck.

The performance evaluation has shown that numerical programs are limited by the memory system, while multimedia programs are not. Although the introduction of a real memory system has strongly reduced the performance values obtained by the SSV architecture, these performance results are still higher than the SS performance values.

The study inside regions has presented that, for the SSV architecture, programs scale well inside D-regions and behave constant inside S-regions. The overall performance is determined by the weight of the S- and D-regions in the whole program. Inside S-regions the SS architecture reaches a better performance due to the scaling of the superscalar core.

All in all, the CB memory model presents better performance results for the numerical programs and the difference with the CA performance results for multimedia programs is small. Therefore, the CB memory model will be the memory model that we propose to be attached to the SSV processor, as it achieves the better results both, in the cache efficiency and performance studies.

We conclude that the SSV architecture is a feasible architecture from the performance point of view. It reaches a better performance than a traditional SS architecture, either with ideal or real memory systems, as the processor configuration is scaled. It is a good choice for multimedia programs and it achieves really good results for numerical programs. These results can still be improved by tuning the memory hierarchy.

Regarding the size of the vector length to be used, we observe that, although, in general, the better performance results are obtained by using 128-element vector registers, the final decision depends on the available transistor budget for the concrete design. An implementation for a general purpose processor should use 128-element vector registers in order to achieve the better performance. However, a low cost approach can sacrifice some of the obtained performance in order to decrease its cost, thus including 16-element vector registers.

Tuning the Memory Hierarchy

The study under a real cache hierarchy evidenced some bottlenecks related to the memory hierarchy design. These bottlenecks mainly appeared in numerical programs with either large working sets or strided memory accesses. These programs had a large amount of traffic with the main memory, a relatively low hit rate, and a vector cache stall time that in some cases reached up to 60% of the total execution time.

In order to decrease the effect of these bottlenecks and improve the performance of the SSV architecture we have proposed several enhancements.

The first enhancement to the baseline SSV architecture plus CB memory model under study has been to add additional Miss Status Holding Registers and Write Buffer

entries. Increasing the memory non-blocking mechanisms in the SSV architecture diminishes the negative effect of the vector cache stall. The performance improvements obtained with this tuning reaches up to 37% for 128-element vector registers (46% for 64-element vector registers).

We also studied the possibility of adding an extra scalar memory port to the SSV architecture, as a key limiting resource in the SSV architecture can be the number of scalar memory references it can perform in parallel (1 scalar reference per cycle at most). Results showed that adding an extra port for scalar accesses is a profitable enhancement for running a heterogeneous set of programs, that could even include low vectorization programs, in the SSV, as it provides with some additional flexibility in accessing memory.

Another tuning that we studied has been the increase of the memory bandwidth. This tuning has been motivated by the large amount of traffic that numerical programs generated to/from the RDRAM array. For that reason these programs have taken the better advantage of doubling main memory bandwidth. As we increase the RDRAM bandwidth in the SSV-4 architecture, programs limited by memory bandwidth increase their performance by up to 20%.

Finally, we also studied in this chapter two possible future evolutions of the vector cache. The first one consists in increasing its size, associativity and latency, at the same time that the memory bandwidth is also increased. This analysis has shown that programs that are constrained by the memory hierarchy improve their performance as the L2 cache parameters are improved. Meanwhile, the rest of the programs suffer a performance loss because of the prevalence of the higher latency over the reduced main memory traffic.

The second future enhancement of the vector cache consists in using the extra on-chip space in order to implement more complex cache designs, aimed at dealing with the non stride-1 memory references found in both numerical and multimedia programs. Results showed that these new designs, called Collapsing Vector Cache and Multi-

Address cache, deliver improvements with respect to the base case that range from 5% up to 128%.

The Collapsing Vector Cache (CVC) uses a collapsing buffer that is able to retrieve several vector elements along two consecutive cache lines, even if they are not consecutively allocated. It improves performance in almost all programs, from 5% to 90% for both 128-element and 64-element vector lengths. Improvements are concentrated in programs that have strides 2, 3 and 4, which benefit from the collapsing hardware.

The Multi-Address Cache (MAC) is a conventional cache where a vector memory access is decoupled among all available memory ports. As we can send independent memory addresses no matter the stride between them, this alternative is more flexible and expensive to implement. MAC obtains better performance results, even though it has a larger cache latency, due to its more flexible design. The larger improvements, which are as large as 128%, happen in programs that either have very large strides, or are strongly memory limited.

Therefore, large performance improvements can be obtained as the cache design is tuned in order to tackle the non stride-1 memory accesses. Although these cache designs are complex and expensive to implement, the large performance improvements obtained pay off the effort, so that as technology evolves they can actually be implemented.

7.2 FUTURE WORK

Implementation cost of the ILP+DLP architecture

The performance study carried out in this thesis has revealed that the ILP+DLP architecture reaches good performance results for numerical and multimedia programs. This work could be extended by studying the implementation cost of the ILP+DLP processor through an accurate model that allows calculating the amount of chip area that is required in order to build it. It would be interesting comparing, in terms of cost, the different configurations that have been studied. In this way we could analyze

which, among all the configurations studied, is the configuration with a minimum cost, a minimum power consumption and a better price performance ratio.

Improving gather/scatter and non stride-1 performance

We have observed in this work that non stride-1 vector memory accesses, as well as gather/scatter vector memory accesses do not have enough spatial locality and, consequently, do not perform well under a traditional memory hierarchy. Although we have proposed two additional cache hierarchies aimed at dealing with these types of memory accesses, they are costly to implement. It would be interesting to study additional memory models that could improve performance of gather/scatter and non stride-1 vector memory accesses while keeping a low cost. One solution could be not to bring to the cache these types of accesses, but rather access them directly to the main memory. Although the latency would be higher, these data would not pollute the cache causing conflicts and evicting useful data.

Understanding how a VLIW performs on D-regions

An alternative to using our proposed vector extensions to attack the parallelism present in D-regions is the use of statically-scheduled VLIW architecture. It would be interesting to compare the relative merits of VLIW vs. vector on these highly regular and data-parallel sections of code to understand which paradigm provides a better performance and better price-performance rate.

Low End ILP+DLP processors

In this thesis we have focused on how can we use the additional transistors that will be available on a chip in the near future. The goal was to achieve the better performance by using as many transistor as it could be possible. However, it is also interesting to study the low end spectrum of ILP+DLP processors, which can yield good performance at a lower cost. In this sense, we could study a modest ILP processor coupled with a

vector unit that uses short vector registers. These processors will be aimed at a low power consumption and dealing with the problem of the wire delays.

In-order versus Out-of-order ILP+DLP processors

Also related with the previous topic it would be interesting to extend this work in order to study the relative performance behaviors of the ILP core of the ILP+DLP architecture whenever this core carries out in-order versus out-of-order execution. Although the in-order ILP core is much more limited than the out-of-order core that we have studied, its cost would be also lower, so that the price/performance ratio would be well-suited for the execution of applications with a reasonable performance and low cost.

Simple Simultaneous Multithreaded ILP+DLP processors

As discussed earlier, different techniques aimed at the exploitation of different styles of parallelism are currently being merged into a single processor. It would be interesting to study the performance of a simultaneous multithreaded ILP+DLP architecture backed with a memory hierarchy based on a vector cache. In order to restrict the study to a feasible set of configurations, the study would be focused on a narrow-way ILP core, with a vector functional unit with short vectors, and a modest simultaneous multithreaded execution, running at a high clock rate.

LIST OF FIGURES

Chapter 4

Chapter 5

Chapter 6

LIST OF TABLES

REFERENCES

[AA93] D. Alpert and D. Avnon. Architecture of the Pentium microprocessor. *IEEE Micro*, 13(3):11–21, 1993.

[AAD+93] Tom Asprey, Gregory S. Averill, Eric DeLano, Russ Mason, Bill Weiner, and Jeff Yetter. Performance Features of the PA7100 Microprocessor. *IEEE Micro*, pages 22–35, June 1993.

[AHKB00] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *27th International Symposium on Computer Architecture*, pages 248–259, 2000. Available at http://citeseer.nj.nec.com/agarwal00clock.html.

[AJ88] Randy Allen and Steven C. Johnson. Compiling C for vectorization, parallelization, and inline expansion. Technical report, 1988.

[Amd67] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings AFIPS 1967 Spring Joint Computer Conference*, pages 483–485, April 1967.

[Arn83] C. Arnold. Vector Optimization on the Cyber 205. In *International Conference on Parallel Processing*, August 1983.

[BA97] D. Burger and T. Austin. The SimpleScalar tool set version 2.0., June 1997.

[Ban98] P. Bannon. Alpha 21364: A Scalable Single-chip SMP, 1998.

[BCK+89] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Scheider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl,

O. Johnson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989. Available at http://citeseer.nj.nec.com/berry88perfect.html.

[BDHS94] Jama Barreh, Sudhir Dhawan, Troy Hicks, and David Shippy. The POWER2 Processor. In *COMPCON*, 1994.

[BE98] J. Bier and J. Eyre. Independent DSP benchmarking: methodologies and latest results. In *International Conference on Signal Processing Applications and Technology*, September 1998.

[BGB98] L. Barroso, K. Gharachorloo, and F. Bugnion. Memory System Characterization of Commercial Workloads. In *25th ISCA*, pages 3–14, 1998.

[BGK96] Doug Burger, James R. Goodman, and Alain Kägi. Memory Bandwidth Limitations of Future Microprocessors. In *23th International Symposium on Computer Architecture*, pages 78–89, 1996.

[BGM+00] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *25th Annual International Symposium on Computer Architecture*, pages 282–293, 2000.

[BK95] P. Bannon and J. Keller. Internal architecture of Alpha 21164 microprocessor, 1995.

[Bos88] Pradip Bose. Heuristic, rule-based program transformations for enhanced vectorization. *PROC of the 1988 ICPP*, II, Software:63–66, August 1988. IBM TJW.

[Bra86] Thomas Brandes. Automatic vectorization for high level languages based on an expert system. *CONPAR 86, CONF on Algorithms and Hardware for Parallel Processing (LNCS 237)*, pages 303–310, September 1986.

[BS00] V. Bongiorno and G. Shorrel. Cray SV1, SV1e, SV1ex - Overview, 2000. Available at http://www.cray.com/products/systems/sv1.

[Bud84] Timothy A. Budd. An APL compiler for a vector processor. *TOPLAS*, 6(3):297–313, July 1984.

[CB92] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 27, pages 51–61, New York, NY, 1992. ACM Press. Available at http://citeseer.nj.nec.com/chen92reducing.html.

[CB96] Zarka Cvetanovic and Dileep Bhandarkar. Performance Characterization of the Alpha 21164 Microprocessor Using TP and SPEC Workloads. In *Second International Symposium on High-Performance Computer Architecture*, pages 270–280, Los Alamitos, California, 1996. IEEE Computer Society Press.

[CEV98] Jesus Corbal, Roger Espasa, and Mateo Valero. Command vector memory systems: High performance at low cost. In *pact*, October 1998.

[CEV99] Jesus Corbal, Roger Espasa, and Mateo Valero. Exploiting a New Level of DLP in Multimedia Applications. In *32th International Symposium on Microarchitecture*, November 1999.

[CEV01] Jesus Corbal, Roger Espasa, and Mateo Valero. On the Efficiency of Reductions in micro-SIMD Media Extensions. In *International Conference on Parallel Architectures and Compilation Tecniques, PACT-01, Barcelona*, September 2001.

[CGMW88] Mike Chastain, Gary Gostin, Jim Mankovich, and Steve Wallach. The CONVEX C240 architecture. In *Proceedings of Supercomputing'88*, pages 321–329, Orlando, Florida, November 1988. IEEE Computer Society Press.

[CGVT00] Jose-Lorenzo Cruz, Antonio Gonzalez, Mateo Valero, and Nigel P. Topham. Multiple-banked register file architectures. In *27th International Symposium on Computer Architecture*, pages 316–325, 2000.

[Chr96] Dave Christie. Developing the AMD-K5 Architecture. *IEEE Micro*, pages 16–26, April 1996.

[CKDK91] Robert F. Cmelik, Shing I. Kong, David R. Ditzel, and Edmund J. Kelly. An analysis of SPARC and MIPS instruction set utilization on the SPEC benchmarks. In *4th International Conference on Architectural Support*

for Programming Languages and Operating Systems, pages 290–302, Santa Clara, California, 1991.

[CKPK90] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer Performance Evaluation and the Perfect Benchmarks. In *International Conference on Supercomputing*, 1990.

[CMMP95] Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, and Burzin A. Patel. Optimization of Instruction Fetch Mechanisms for High Issue Rates. In *ISCA 22*, pages 333–344, 1995.

[Con92] Convex Press, Richardson, Texas, U.S.A. *CONVEX Architecture Reference Manual (C Series)*, sixth edition, April 1992.

[Con93] Convex Computer Corporation. *Saturn Architecture Specification, Revision 1.4*, April 1993.

[Cri97] Richard Crisp. Direct rambus technology: The new main memory standard. *IEEE Micro*, pages 18–28, November 1997.

[CTS96] Chih-Yung Chang, Jiann-Yuann Tzeng, and Jang-Ping Sheu. Design and Implementation of a Fortran Assistant Tool for Vector Compilers. *Intl. Journal of High Speed Computing*, 8(1):13–45, 1996.

[DD97] Keith Diefendorff and Pradeep K. Dubey. How Multimedia Workloads Will Change Processor Design. *Computer*, 30(9):43–45, 1997.

[Del91] G. Delic. Performance analysis of a 24 code sample on Cray X/Y-MP at the Ohio Supercomputer Center. In *Proceedings of the 5th SIAM Conference on Parallel Processing for Scientific Applications*, pages 530–535, 1991.

[DF90] P. Dubey and M. Flynn. Optimal pipelining, 1990.

[DH] J. J. Dongarra and A. Hinds. Comparison of the CRAY X-MP-4, the Fujitsu VP-200, and the Hitachi S-810/20: an Argonne perspective. Technical report, Argonne National Laboratory.

[Die99] Keith Diefendorff. Power4 focuses on memory bandwidth. *Microprocessor Report*, 13(13), October 1999.

[Don93] J. J. Dongarra. Performance of various computers using standard linear equations software in a fortran environment. Technical Report CS-89-85, Univeristy of Tennesse, 1993.

[DT92] H. Dwyer and H. C. Torng. An out-of-order superscalar processor with speculative execution and fast, precise interrupts. In *25th Annual International Symposium on Microarchitecture*, pages 272–281, Portland, Oregon, 1992. ACM and IEEE.

[EGK+94] Kemal Ebcioglu, Randy D. Groves, Ki-Chang Kim, Gabriel M. Silberman, and Isaac Ziv. VLIW Compilation Techniques in a Superscalar Environment. In *Programming Languages Design and Implementation*, pages 36–48, 1994.

[EM94] Roger Espasa and Xavier Martorell. Dixie: a trace generation system for the C3480. Technical Report CEPBA-RR-94-08, Universitat Politècnica de Catalunya, 1994.

[Eme99] J. Emer. Simultaneous multithreading: Multiplying alpha's performance, October 1999.

[Esp95] Roger Espasa. JINKS: A parametrizable simulator for vector architectures. Technical Report UPC-CEPBA-1995-31, Universitat Politècnica de Catalunya, 1995.

[Esp97] Roger Espasa. *Advanced Vector Architectures.* PhD thesis, Universidad Politecnica de Catalunya, February 1997.

[EV97] Roger Espasa and Mateo Valero. Exploiting Instruction- and Data-Level Parallelism. *IEEE Micro*, 17(5):20–??, /1997. Available at http://citeseer.nj.nec.com/espasa97exploiting.html.

[EVS97] Roger Espasa, Mateo Valero, and James E. Smith. Out-of-order Vector Architectures. In *MICRO-30*, pages 160–170. IEEE Press, December 1997.

[EVS98] R. Espasa, M. Valero, and J. Smith. Vector Architectures: Past, Present and Future (Keynote Speech). In *12th International Conference on Supercomputing*, pages 425–432, June 1998.

[FJC96] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. Register File Design Considerations in Dynamically Scheduled Processors. In *Proceedings of the Second International Symposium on High Performance Computer Architecture*, January 1996.

[FLTB99] C. Frouzakis, J. Lee, A. Tomboulides, and K. Boulouchos. Two-Dimensional Direct Numerical simulation of Opposed-Jet Hydrogen/Air Difusion Flame, 1999.

[Fly97] M. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design.* Jones and Bartlett, 1997.

[FS94] Manoj Franklin and Mark Smotherman. A Fill-Unit Approach to Multiple Instruction Issue. In *27th International Symposium on Microarchitecture*, pages 162–171, 1994.

[Fuj01] Fujitsu. High Performance Computers: VPP5000, 2001. Available at http://www.fse.fujitsu.com/products/vpp5000.html.

[Gas89] F. Gasperoni. *Compilation techniques for VLIW architectures.* IBM Research Division, T. J. Watson Research Center, 1989.

[Gas91] F. Gasperoni. *Scheduling for Horizontal Systems: The VLIW Paradigm in Persepctive.* PhD thesis, May 1991. Available at http://citeseer.nj.nec.com/gasperoni91scheduling.html.

[GBH96] Ralf Gruber, James Brunson, and Michael Hodous. Preliminary Benchmark Measurments on the NEC SX-4. *SPEEDUP Journal*, 9(2):76–79, 1996.

[GBL+98] J. Gibson, T. Berger, T. Lookabaugh, D. Lindbergh, and R. Baker. *Digital Compression for Multimedia.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1998.

[GGV98] A. Gonzalez, J. Gonzalez, and M. Valero. Virtual-Physical Registers, 1998.

[Gie97] B. Gieseke. A 600MHz Superscalar RISC Microprocessor with Out-of-Order Execution, 1997.

[GM96] F. Gabbay and A. Mendelson. Speculative Execution based on Value Prediction. Technical Report EE Department TR-1080, Technion–Israel Institute of Technology, Israel, 1996. Available at http://citeseer.nj.nec.com/gabbay96speculative.html.

[Hil87] Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* PhD thesis, University of California at Berkeley Computer Science Division, 1987. Tech. Rep. UCB/CSD 87/381 (November).

[Hil88] Mark D. Hill. A Case for Direct Mapped Caches. *IEEE Computer*, 21(12):25–40, December 1988.

[Him96] R. Himeno. Car Aerodynamics Simulation on NEC SX-4, 1996. Available at http://www.sw.nec.co.jp/hpc/sx-e/sx-world.

[Him00] R. Himeno. Computational Biomechanics Research Project Started at RIKEN, 2000. Available at http://www.sw.nec.co.jp/hpc/sx-e/sx-world.

[HKLS00] D. Henry, B. Kuszmaul, G. Loh, and R. Sami. Circuits for Wide-Window Superscalar Processors. In *27th International Symposium on Computer Architecture*, pages 236–247, 2000.

[HS93] W. C. Hsu and J. E. Smith. Performance of cached DRAM organizations in vector supercomputers. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 327–336, San Diego, California, May 17–19, 1993. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 21(2), May 1993.

[Hsu94] P. Hsu. Design of the R8000 Microprocessor, April 1994.

[HT72] R. G. Hintz and D. P. Tate. Control data STAR-100 processor design. In *Proc. Compcon 72*, pages 1–4, New York, 1972. IEEE Computer Society Conf. 1972, IEEE.

[Hun95] Doug Hunt. Advanced Performance Features of the 64-bit PA-8000. In *COMPCON*, 1995.

[IKFN98] M. Ionue, H. Katayama, T. Furui, and K. Nakanishi. Development Concepts and Overview of the SX-5 Series Supercomputers, 1998. Available at http://www.sw.nec.co.jp/hpc/sx-e/sx-world/.

[Iko00] T. Ikohagi. Grappling with Numerical Simulation of Complex Fluid Phenomenon, 2000. Available at http://www.sw.nec.co.jp/hpc/sx-e/sx-world.

[Inc01] Cray Inc. Cray SV2, 2001. Available at http://www.cray.com/products/systems/sv2.

[Int00] Intel Corporation. *A Detailed Look Inside the Intel NetBurst Micro-Architecture of the Intel Pentium 4 Processor. Revision 1.0*, November 2000.

[IW91] Akihiro Iwaya and Tadashi Watanabe. The parallel processing feature of the NEC SX-3 supercomputer system. *Intl. Journal of High Speed Computing*, 3(3&4):187–197, 1991.

[JNT97] Toni Juan, Juan J. Navarro, and Olivier Temam. Data Caches for Superscalar Processors. In *International Conference on Supercomputing*, pages 60–67, 1997. Available at http://citeseer.nj.nec.com/juan97data.html.

[Joh91] M. Johnson. *Superscalar Microprocessor Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[JRB+98] Stephan Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. A Novel Renaming Scheme to Exploit Value Temporal Locality Through Physical Register Reuse and Unification. In *International Symposium on Microarchitecture*, pages 216–225, 1998.

[JSN98] Toni Juan, Sanji Sanjeevan, and Juan J. Navarro. Dynamic History-length Fitting: A Third Level of Adaptivity for Branch Prediction. In *25th International Symposium on Computer Architecture*, pages 155–166, 1998.

[JTVW01] Ben Juurlink, Dmitri Tcheressiz, Stamatis Vassiliadis, and Harry Wijshoff. Implementation and Evaluation of the Complex Streamed Instruction Set. In *International Conference on Parallel Architectures and Compilation Tecniques, PACT-01, Barcelona*, pages 73–82, September 2001.

[JW94] Norman P. Jouppi and Steven J. E. Wilton. Tradeoffs in Two-Level On-Chip Caching. In *21th International Symposium on Computer Architecture*, pages 34–45, 1994.

[Kah99] J. Kahle. Power4: A Dual-CPU Processor Chip, October 1999.

[Kan97] M. Kanda. Cerius/ADF: Density Functional Method Molecular Orbital Program, 1997. Available at http://www.sw.nec.co.jp/hpc/sx-e/sx-world.

[Kat00] M. Kato. PAM-CEM: the Next-Generation CEM Simulation Software, 2000. Available at http://www.sw.nec.co.jp/hpc/sx-e/sx-world.

[Kel96] Jim Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution. In *Microprocessor Forum*, October 1996.

[KELS62] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level Storage system. *IRE Transactions on Electronic Computers*, EC-11:223–235, April 1962. Also appears in [SBN82, pages 135–148].

[Kes99] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, /1999.

[KIS+94] Katsuyoshi Kitai, Tadaaki Isobe, Tadayuki Sakakibara, Shigeko Yazawa, Yoshiko Tamaki, Teruo, and Kouichi Ishii. Distributed storage control unit for the Hitachi S-3800 multivector supercomputer. In *Proceedings of the International Conference on Supercomputing*, pages 1–10, July 1994.

[KM89] Les Kohn and Neal Margulis. Introducing the Intel i860 64-bit microprocessor. *IEEE Micro*, 9(4):15–30, 1989.

[Kog81] P. Kogge. *The Architecture of Pipelined Computers.* McGraw Hill Book Company, New York, NY, 1981.

[Koh95] L. Kohn. The Visual Instruction Set (VIS) in UltraSPARC. In *40th Annual Compcon*, 1995.

[Kro81] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *The 8th Annual International Symposium on Computer Architecture*, pages 81–87, 1981.

[KSF+94] L.I. Kontothanassis, R. A. Sugumar, G. J. Faanes, J. E. Smith, and M. L. Scott. Cache performance in vector supercomputers. In *Proceedings of Supercomputing'94*, Washington D.C., November 1994. IEEE Computer Society Press.

[Kum96] A. Kumar. The HP-PA8000 RISC CPU: A High Performance Out-of-Order Processor. In *Hot Chips VIII*, pages 9–20, August 1996.

[Lam88] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI)*, volume 23, pages 318–328, 1988.

[Lan98] C. Lantwin. The 10 Past and 10 Future Years of HPC: A Panel Discussion, 1998. Available at http://www.sw.nec.co.jp/hpc/sx-e/sx-world/.

[Lan00] C. Lantwin. A Crossroads to Future, 2000. Available at http://www.sw.nec.co.jp/hpc/sx-e/sx-world/.

[LBCG95] D. Lee, J. Baer, B. Calder, and D. Grunwald. Instruction Cache Fetch Policies for Speculative Execution. In *22th International Symposium on Computer Architecture*, pages 357–367, May 1995.

[LD97] C. Lee and D. DeVries. Initial results on the performance and cost of vector microprocessors. In *30th Annual International Symposium on Microarchitecture*, pages 171–182, December 1997.

[Lee96] Ruby Lee. Subword Parallelism with MAX–2, 1996.

[LL00] C. Lantwin and C. Lazou. Weather Forecast in Europe, 2000. Available at http://www.sw.nec.co.jp/hpc/sx-e/sx-world.

[LPMS97] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *30th International Symposium on Microarchitecture*, pages 330–335, December 1997. Available at http://www.cs.ucla.edu/̃leec/mediabench/.

[LR91] M.S. Lam and M.C. Rinard. Coarse-grain parallel programming in jade. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 26, pages 94–105, Williamsburg, VA, April 1991.

[LS98] C. Lee and M. Stoodley. Simple Vector Microprocessors for Multimedia Applications. In *31st Annual International Symposium on Microarchitecture*, pages 330–335, December 1998.

[LW92] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Nineteenth International Symposium on Computer Architecture*, pages 46–57, Gold Coast, Australia, 1992. ACM and IEEE Computer Society.

[LW97] Heng Liao and Andrew Wolfe. Available Parallelism in Video Applications. In *Proceeding on the 30th Int. Symp on Microarchitecture*, pages 321–329, Dec. 1997.

[LWS96] M. Lipasti, C. Wilkerson, and J. Shen. Value Locality and Load Value Prediction. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, New York, 1996. ACM Press.

[Mal95] Michael Malone. *The Microprocessor: A Biography.* Springer–Verlag Inc., Telos, Santa Clara, California, 1995.

[Mar98a] D. Maric. Obtaining High Sustained Performance with Real-life Applications at CSCS-ETHZ, 1998.

[Mar98b] K. Maruyama. Global Environment and Climate Projection, 1998. Available at http://www.sw.nec.co.jp/hpc/sx-e/sx-world.

[McF93] Scott McFarling. Combining Branch Predictors. Technical Report TN-36, June 1993. Available at http://citeseer.nj.nec.com/mcfarling93combining.html.

[McL93] E. McLellan. The Alpha AXP Architecture and 21064 Processor. *IEEE Micro*, 13(3):36–47, June 1993.

[Mic01] Microprocessor Report. *Chart Watch: Workstation Processors*, August, 27 2001. Available at http://mdronline.com/.

[MIP97] MIPS Technologies, Inc. *MIPS Extension for Digital Media with 3D*, March 1997. Available at http://www.mips.com/arch/ISA5/.

[NEC98] NEC. Parallelization of Scientific Applications on the SX-4 - The CSCS-NEC, 1998.

[NEC01] NEC Supercomputer Systems. *The NEC SX-6: Supercomputer with Single-Chip Vector Processor*, 2001. Available at http://www.ess.nec.de/newsroom/attachments/SX-6-Single-node.pdf.

[NJ99] H. Nguyen and L. John. Exploiting SIMD parallelism in DSP and multimedia algorithms using the AltiVec technology. In *13th ACM International Conference on Supercomputing*, pages 1120–, June 1999.

[NKT+95] Y. Nakashima, T. Kitamara, H. Tamura, M. Takiuchi, and K. Miura. Scalar processor of the VPP500 parallel supercomputer. In *Proceedings of the Nineth International Conference on Supercomputing*, pages 348–356, July 1995.

[Oed92] W. Oed. Cray Y-MP C90: System features and early benchmark results. *Parallel Computing*, 18(8):947–954, August 1992.

[Pap96] David B. Papworth. Tuning the Pentium Pro Microarchitecture. *IEEE Micro*, pages 8–15, April 1996.

[PAY96] Dave Patterson, Tom Anderson, and Kathy Yelick. A Case for Intelligent DRAM: IRAM. In *Hot Chips VIII*, August 1996.

[PB88] D. V. Pryor and P. J. Burns. Vectorized Monte Carlo Molecular Aerodynamics Simulation of the Reyleigh Problem. In *Proceedings of Supercomputing'88*, pages 384–391, Orlando, Florida, November 1988. IEEE Computer Society Press.

[PH96] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1996.

[PJS96] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Quantifying the Complexity of Superscalar Processors. Technical Report CS-TR-1996-1328, 1996.

[PJS97] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processors. In *International Annual Symposium on Computer Architecture*, pages 206–218, 1997. Available at http://citeseer.nj.nec.com/73821.html.

[PM86] Robert Perron and Craig Mundie. The architecture of the Alliant FX/8 computer. In *Digest of Papers: Compcon 86*, pages 390–394, Washington, D.C., March 1986.

[Pri96] Betty Prince. *High Performance Memories.* Wiley & Sons, Ltd., 1996.

[PSW91] C. Peterson, J. Sutton, and P. Wiley. iWarp: a 100-MOPS, LIW microprocessor for multicomputers. *IEEE Micro*, 11(3), June 1991.

[PVAL95] Montse Peiron, Mateo Valero, Eduard Aygaudé, and Tomás Lang. Vector multiprocessors with arbitrated memory access. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 243–252, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News*, 23(2), May 1994.

[PW94] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line, 1994.

[PW96] Alex Peleg and Uri Weiser. MMX technology extension to the Intel architecture — improving multimedia and communications application performance by 1.5 to 2 times. *IEEE Micro*, 16(4):42–50, August 1996.

[QCEV99] Francisca Quintana, Jesus Corbal, Roger Espasa, and Mateo Valero. Adding a Vector Unit to a Superscalar Processor. In *International Conference on Supercomputing*, pages 333–344. ACM SIGARCH and IEEE Computer Society TCCA, June 1999.

[QCEV01] Francisca Quintana, Jesus Corbal, Roger Espasa, and Mateo Valero. A Cost Effective Architecture for Vectorizable Numerical and Multimedia Applications. In *13th ACM Symposium on Parallel Algorithms and Architectures*, pages 103–112, July 2001.

[QEV98a] Francisca Quintana, Roger Espasa, and Mateo Valero. A Case for Merging the ILP and DLP Paradigms. In *6th Euromicro Workshop on Parallel and Distributed Processing*, pages 217–224. Lecture Notes in Computer Science, Springer-Verlag, January 1998.

[QEV98b] Francisca Quintana, Roger Espasa, and Mateo Valero. An ISA Comparison between Superscalar and Vector Processors. In *International Conference on Vector and Parallel Processing (VECPAR)*. Lecture Notes in Computer Science, Springer-Verlag, June 1998.

[Ran96] H. Rangu. Research in Computational Material Science at the NEC fundamental Research Laboratories, 1996. Available at http://www.sw.nec.co.jp/hpc/sx-e/sx-world.

[Rau93] B. Ramakrishna Rau. Dynamically Scheduled VLIW Processors. In *26th Annual International Symposium on Microarchitecture*, pages 80–92, 1993.

[RBS96] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *International Symposium on Microarchitecture*, pages 24–35, 1996. Available at http://citeseer.nj.nec.com/rotenberg96trace.html.

[RDK+98] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. A Bandwidth-efficient Architecture for Media Processing. In *International Symposium on Microarchitecture*, pages 3–13, 1998. Available at http://citeseer.nj.nec.com/rixner98bandwidthefficient.html.

[RF] Kevin W. Rudd and Michael J. Flynn. Instruction-Level Parallel Processors – Dynamic and Static Scheduling Tradeoffs. Available at http://citeseer.nj.nec.com/211594.html.

[RF93] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing*, 7(1-2):9–50, 1993.

[Rhy99] Lee Rhyne. CFX: General-Purpose Computational Fluid Dynamics Analysis Code, 1999. Available at http://www.sw.nec.co.jp/hpc/sx-e/sx-world.

[Ric78] R. Richard. The Cray-1 Computer System, 1978.

[RLPN+99] Alex Ramirez, Josep-Lluis Larriba-Pey, Carlos Navarro, Josep Torrellas, and Mateo Valero. Software trace cache. In *Proceeding of the 13th Inter-*

national Conference on Supercomputing, pages 119–126, 1999. Available at http://citeseer.nj.nec.com/article/ramirez99software.html.

[RQJS97] E. Rotenberg, Y. Sazeides Q. Jacobson, and Jim Smith. Trace processors. In *30th Annual International Symposium on Microarchitecture*, pages 138–148, December 1997.

[RR94] K.A. Robbins and S Robbins. Relationship between average and real memory behavior. *The Journal of Supercomputing*, 8(3):209–232, November 1994.

[RTAD97] J.A. Rivers, G.S. Tyson, T.M. Austin, and E.S. Davidson. On high-bandwidth data cache design for multiple-issue processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 46–56, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[Rus78] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[SBN82] D. P. Siewiorek, C. G. Bell, and A. Newell. *Computer Structures: Principles and Examples.* Mc-Graw Hill, New York, 1982.

[SC97] Kevin Skadron and Douglas W. Clark. Design issues and tradeoffs for write buffers. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 144–155, San Antonio, Texas, February 1–5, 1997. IEEE Computer Society TCCA.

[SCEV99] Esther Salami, Jesus Corbal, Roger Espasa, and Mateo Valero. An Evaluation of Different DLP alternatives for the Embedded Media Domain. In *1st Workshop on Media Processors and DSPs (MP-DSP), In Conjunction with MICRO-32*, November 1999.

[SDC94] S. P. Song, M. Denman, and J. Chang. The PowerPC 604 RISC Microprocessor. *IEEE Micro*, 14(5):8–17, 1994.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Anal-

ysis Tools. *ACM SIGPLAN Notices*, 29(6):196–205, 1994. Available at http://www.research.digital.com/wrl/publications/abstracts/94.2.html.

[SF91] G. Sohi and M. Franklin. High-Bandwidth Data Memory Systems for Superscalar Procesors. In *4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 26, pages 53–62, New York, NY, 1991. ACM Press.

[SFK97] D. Sima, T. Fountain, and P. Kacsuk. *Advanced Computer Architecture: A design space approach.* Addison-Wesley, Essex, England, 1997. ISBN-0-201-42291-3.

[SFS00] J. E. Smith, Greg Faanes, and Rabin A. Sugumar. Vector instruction set support for conditional operations. In *ISCA*, pages 260–269, 2000.

[SG83] James E. Smith and James R. Goodman. A study of instruction cache organizations and replacement policies. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 132–137, Stockholm, Sweden, June 13–17, 1983. *Computer Architecture News,* 11(3), June 1983.

[SH91] Willi Schönauer and Hartmut Häfner. Supercomputers: Where are the lost cycles ? *Supercomputing*, 1991.

[SH94] Willi Schönauer and Hartmut Häfner. Explaining the gap between theoretical peak performance and real performance for supercomputer architectures. *Scientific Programming*, 3:157–168, 1994.

[Sha99] Harsh Sharangpani. Intel Itanium Processor Microarchitecture Overview, October 1999.

[Sit92] R. Sites. *DEC Alpha Architecture.* Digital Press, Burlington, Massachussets, 1992.

[SJSM96] André Seznec, Stéphan Jourdan, Pascal Sainrat, and Pierre Michaud. Multiple-block ahead branch predictors. *ACM SIGPLAN Notices*, 31(9):116–127, 1996.

[SK86] R. G. Scarborough and H. G. Kolsky. A vectorizing FORTRAN compiler. *IBM Journal of Research and Development*, 30(2), March 1986.

[Sla96] Michael Slater. The microprocessor today. *IEEE Micro*, pages 32–44, December 1996.

[Smi82] James E. Smith. Decoupled Access/Execute Computer Architectures. In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 112–119, 1982.

[Smi91] L. L. Smith. Vectorizing C Compilers: How Good are They ? In *Proceedings of Supercomputing'91*, pages 544–553, Albuquerque, New Mexico, November 1991. IEEE Computer Society Press.

[SP94] Eric Sprangle and Yale Patt. Facilitating Superscalar Processing via a Combined Static/Dynamic Register Renaming Scheme. In *27th International Symposium on Microarchitecture*, pages 143–147, December 1994.

[SPE] SPEC:. The standard performance evaluation corporation. Available at http://www.spec.org/.

[SS95] J. Smith and G. Sohi. The Microarchitecture of SuperScalar Processors, December 1995.

[SV87] G. S. Sohi and S. Vajapayem. Instruction issue logic for high-performance, interruptable pipelined processors. In *14th Annual Symposium on Computer Architecture*, pages 27–36, Pittsburgh, Pennsylvania, 1987. IEEE Computer Society and ACM.

[SV97] Dimitrios Stiliadis and Anujan Varma. Selective Victim Caching: A Method to Improve the performance of Direc-Mapped caches. *IEEE Transactions On Computers*, 46(5):603–610, May 1997.

[SWL+91] Margaret L. Simmons, Harvey J. Wasserman, Olaf M. Lubeck, Christopher Eoyang, Raul Mendez, Hiroo Harada, and Misako Ishiguro. A Performance Comparison of Three Supercomputers: Fujitsu VP-2600, NEC SX-3 and Cray Y-MP. In *Proceedings of Supercomputing'91*, pages 150–157, Albuquerque, New Mexico, November 1991. IEEE Computer Society Press.

[TEL95] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *22th Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[TGN95] M. Tremblay, D. Greenley, and K. Normoyle. The design of the microarchitecture of UltraSPARC-I. *Proceeddings of the IEEE*, 83(12):1653–1663, December 1995.

[TLO98] A. Tomboulides, J. Lee, and S. Orszag. Simulation of low mach number reactive flows: algorithm, analysis and applications, 1998.

[Tom67] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, January 1967.

[TONH96] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. VIS Speeds New Media Processing. *IEEE Micro*, pages 10–20, August 1996.

[Tsu99] M. Tsukada. Computational Physics for Surfaces and Interfaces - Using the Sx-4B/2A, 1999. Available at http://www.sw.nec.co.jp/hpc/sx-e/sx-world.

[Uch97] N. Uchida. Hardware of the VX/VPP300/VPP700 Series of Vector-Parallel Computer Systems, 1997.

[vdSD01a] A. van der Steen and J. Dongarra. The Cray Inc. SV1ex, August 2001. Available at http://www.top500.org/ORSC/2001.

[vdSD01b] A. van der Steen and J. Dongarra. The Fujitsu VPP5000 Series, August 2001. Available at http://www.top500.org/ORSC/2001.

[vdSD01c] A. van der Steen and J. Dongarra. The NEC SX-5, August 2001. Available at http://www.top500.org/ORSC/2001.

[Vei97] A. V. Veidenbaum. Instruction Cache Prefetching Using Multilevel Branch Prediction. *Lecture Notes in Computer Science*, 1336:51–??, 1997. Available at http://citeseer.nj.nec.com/veidenbaum97instruction.html.

[VEV97] Luis Villa, Roger Espasa, and Mateo Valero. Effective Usage of Vector Registers in Advanced Vector Architectures. In *Parallel Architectures and Compilation Techniques (PACT'97)*, pages 250–260, San Francisco, CA, USA, November 1997. IEEE Computer Society Press. Avaliable at http://citeseer.nj.nec.com/villa97effective.html.

[VEV98] Luis Villa, Roger Espasa, and Mateo Valero. A Performance Study of Out-of-Order Vector Architectures and Short Registers. In *Proceedings of the International Conference on Supercomputing.* ACM Press, July 1998.

[VLL+92] Mateo Valero, Tomás Lang, José M. Llabería, Montse Peiron, Eduard Ayguadé, and Juan J. Navarro. Increasing the number of strides for conflict-free vector access. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 372–381, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 20(2), May 1992.

[VLPA95] Mateo Valero, Tomas Lang, Montse Peiron, and Eduard Ayguade. Conflict-Free Access for Streams in Multimodule Memories. *IEEE Transactions on Computers*, 44(5):634–646, 1995. Available at http://citeseer.nj.nec.com/398945.html.

[VM97] S. Vajapeyam and T. Mitra. Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences. In *International Symposium on Computer Architecture*, pages 1–12, June 1997.

[VSH91] Sriram Vajapeyam, Gurindar S. Sohi, and Wei-Chung Hsu. An empirical study of the Cray Y-MP processor using the PERFECT Club benchmarks. *International Symposium on Computer Architecture*, pages 170–179, 1991.

[Wal91] D. W. Wall. Limits of Instruction-Level Parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 26, pages 176–189, New York, NY, 1991. ACM Press.

[Wat72] W. Watson. The TI-ASC, A highly modular and flexible super computer architecture. *Proc. AFIPS*, 41, pt. 1:221–228, 1972.

[WB96] Steven Wallace and Nader Bagherzadeh. A Scalable Register File Architecture for Dynamically Scheduled Processors. In *Parallel Architectures and Compilation Techniques*, October 1996.

[Wil65] Maurice Wilkes. Slave Memories and Dynamic Storage Allocation. *IRE Transactions on Electronic Computers*, EC-14(2):270–271, April 1965.

[Wil98] William J. Dally. Tomorrow's computing engines (Keynote Speech). In *Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[WKI86] T. Watanabe, H. Katayama, and A. Iwaya. Introduction of the NEC Supercomputer SX System. In Sidney Fernbach, editor, *Supercomputers – Class VI Systems, Hardware and Software*, pages 153–167. North-Holland, Amsterdam, Netherlands, 1986.

[WO95] Kenneth M. Wilson and Kunle Olukotun. High Performance Cache Architectures to Support Dynamic Superscalar Microprocessors. Technical Report CSL-TR-95-682, Stanford University, Computer Systems Lab, 1995.

[WO97] Kenneth M. Wilson and Kunle Olukotun. Designing High Bandwidth On-Chip Caches. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 121–132, June 1997.

[Wol90] Michael Wolfe. *Optimizing supercompilers for supercomputers.* The MIT Press, Cambridge, MA, 1990.

[WOR96] Kenneth M. Wilson, Kunle Olukotun, and Mendel Rosenblum. Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, pages 147–157, 1996.

[WPS95] Ted Williams, Niteen Patkar, and Gene Shen. SPARC64: A 64-b 64-Active-Instruction Out-of-Order-Execution MCM Processor. *IEEE Journal of Solid-State Circuits*, 30(11), November 1995.

[Yag96] Keneth C. Yager. The Mips R10000 Superscalar Microprocessor. *IEEE Micro*, pages 28–40, April 1996.

[YMP93] Tse-Yu Yeh, Deborah T. Marr, and Yale N. Patt. Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. In *International Conference on Supercomputing*, pages 67–76, 1993.

[Yu96] Albert Yu. The future of microprocessors. *IEEE Micro*, pages 46–53, December 1996.

[ZC91] H. Zima and B. Chapman. *Supercompilers for parallel and vector computers.* ACM Press, New York, NY, 1991.

[Zha96] Q. Zhao. *Performance evaluation of instruction prefetching using multilevel branch prediction.* October 1996.

APPENDIX A

Resumen de la tesis (en castellano)

Summary

En cumplimiento del Reglamento de los Estudios de Doctorado de la Universidad de Las Palmas de Gran Canaria, capítulo III, sección 1, artículo 68, este apéndice es un resumen de la tesis doctoral, en castellano, donde se incluyen el objeto y los objetivos de la investigación, el planteamiento y la metodología utilizada, las principales aportaciones y la conclusiones y el trabajo futuro.

A.1 OBJETO Y OBJETIVOS DE LA INVESTIGACIÓN

En las ultimas décadas, los microprocesadores han experimentado un aumento continuo de rendimiento y una reducción en la relación precio/rendimiento [Sla96]. Entre las diferentes tendencias observadas, los procesadores superescalares se han caracterizado por su éxito en diferentes mercados [Sit92] [BDHS94] [Hun95] [TGN95] [WPS95] [Yag96] [Chr96] [Kel96] [Pap96]. A partir de los primeros microprocesadores superescalares, que podían lanzar a ejecutar dos instrucciones en cada ciclo de reloj (HP PA7100 [AAD+93], DEC Alpha [Sit92], Intel Pentium [AA93], IBM PowerPC [SDC94]), se han realizado numerosos trabajos de investigación con el objetivo de mejorar el rendimiento de los mismos. Todos estos trabajos tratan sobre temas como las caches de instrucciones, la predicción de saltos, la organización del banco de registros, el renombre de registros, la posibilidad de lanzar a ejecutar cada vez más instrucciones en cada ciclo de reloj, la planificación dinámica de instrucciones y la ejecución fuera de orden, y las caches de datos (incluyéndose en este último tema las caches multinivel, la prebúsqueda de datos, los accesos no bloqueantes y las caches con múltiples puertos, entre otros).

Todas estas mejoras han permitido avanzar en el camino de la mejora del rendimiento. Los procesadores actuales más avanzados son capaces de operar a frecuencias por encima de 1GHz [Int00] [Kah99] y ofrecen un núcleo de ejecución superescalar, predictores de saltos sofisticados y soporte para sistemas de memoria de alto rendimiento [Int00] [Kah99] [Kes99] [Yag96] [Kum96]. Estos avances, sin embargo, no se producen de manera gratuita, y todas estas técnicas se están complicando cada vez más, como puede verse al examinar las complejas caches de instrucciones de hoy en día [RQJS97] o las técnicas de especulación de datos [GM96] que actualmente se estudian.

Dado que a medida que la tecnología evoluciona se incluirá cada vez un mayor número de transistores en un único chip [Yu96], la cuestión es cómo usarán los procesadores futuros ese incremento en el número de transistores. La mayoría de los procesadores actuales están centrados en el uso de técnicas dirigidas a explotar cada vez más paralelismo a nivel de instrucción [Eme99] [Kah99]. Sin embargo, las medidas del rendimiento real de las aplicaciones que ese ejecutan en estas máquinas [CB96] muestran que el rendimiento alcanzado está muy por debajo del rendimiento teórico máximo de estas

máquinas. Muchos estudios han apuntado que esta falta de rendimiento puede deberse a diferentes efectos, como los fallos en las caches de datos y de instrucciones, los fallos en la predicción de saltos, las dependencias de memoria o la falta de paralelismo de los propios programas [Wal91] [LW92] [BGB98].

Por lo tanto, aunque la explotación del paralelismo a nivel de instrucción ha producido grandes mejoras en el rendimiento, y escalar los procesadores superescalares actuales es un área de investigación muy activa, existe un consenso generalizado en la idea de que este escalado no puede hacerse simplemente buscando, decodificando y ejecutando más y más instrucciones por ciclo de reloj [PJS96] [PJS97] [AHKB00]. Algunas de las razones son las siguientes:

- En primer lugar, es necesario diseñar unidades de búsqueda y decodificación agresivas, lo cual no es un tema trivial debido a la existencia de los saltos, así como el restringido ancho de banda de las caches de instrucciones [PW94] [RBS96] [CMMP95] [RLPN+99].
- En segundo lugar, se necesita una unidad de lanzamiento de instrucciones a ejecución que sea agresiva, con una gran ventana de instrucciones, para así poder alimentar a un gran número de unidades funcionales [HKLS00]. El tiempo de búsqueda en la ventana de instrucciones aumenta cuadráticamente con el tamaño de la misma [PJS97].
- En tercer lugar, para lanzar a ejecutar un gran número de instrucciones se necesita disponer de un banco de registros con gran número de puertos, lo que puede hacer que el tiempo de ciclo del procesador aumente, y además requiere un aumento significativo del área del chip dedicado al banco de registros [CGVT00].
- Por último, realizar múltiples accesos a memoria en cada ciclo de reloj requiere tanto una cache como un TLB multipuerto, y su coste es también proporcional al número de puertos independientes de memoria que posean [JNT97] [BGK96].

Todos estos aspectos hacen que la escalabilidad de los procesadores superescalares sea cara y fuertemente dependiente de la tecnología [AHKB00]. Además, incluso si se

pudiesen resolver estos problemas tecnológicos y de coste con la futura tecnología, veremos a lo largo de esta tesis que el rendimiento obtenido generalmente no justifica la cantidad de área del chip y de esfuerzo de diseño que se requieren [LWS96] [QCEV99].

Por lo tanto, para poder superar los problemas de escalabilidad de los procesadores superescalares actuales, creemos que el camino debe ser la explotación de más de una fuente de paralelismo. Analizaremos primero, en las siguientes secciones, las diferentes fuentes de paralelismo disponibles en los programas, y como se están explotando en los procesadores actuales.

A.1.1 Fuentes de paralelismo

En los programas existen diferentes fuentes de paralelismo: paralelismo de instrucción, paralelismo de datos y paralelismo de thread. Analicemos cada uno de ellos en más detalle.

Paralelismo de instrucción

Entre las diferentes fuentes de paralelismo existentes, el paralelismo de instrucción (ILP) ha sido una de las fuentes de paralelismo más explotadas [RF93]. Un programa presenta ILP cuando diferentes instrucciones de un único flujo de control se pueden ejecutar en paralelo y el resultado final del programa no se altera. La detección de las instrucciones que pueden ser ejecutadas en paralelo puede realizarse en tiempo de compilación o en tiempo de ejecución. Cuando se detectan en tiempo de compilación, el compilador selecciona grupos de instrucciones que pueden ejecutarse en paralelo [Gas89]. Estas instrucciones son típicamente empaquetadas en una única instrucción. Las arquitecturas que explotan ILP de esta manera se denominan arquitecturas de palabra de instrucción muy ancha (Very Long Instruction Word - VLIW) debido a la gran longitud de las instrucciones, que incluyen la especificación de múltiples operaciones [KM89] [PSW91] [Gas91].

El procesador Itanium de Intel-HP [Sha99] es un ejemplo de procesador VLIW. Posee una descripción de paralelismo en el propio diseño, lo que evita que el hardware deba

buscar un paralelismo que el compilador ya conocía. El concepto básico del VLIW es hacer visibles al compilador las decisiones de planificación de instrucciones del hardware. De este modo, el compilador puede hacer las optimizaciones oportunas, puesto que conoce el hardware subyacente donde se van a ejecutar los programas [Lam88]. Los procesadores VLIW también emplean nuevas técnicas [EGK+94] [Rau93], como la ejecución predicada o la especulación, lo que les permite anticiparse, y realizar cálculos antes de que sean necesarios. Por supuesto, el inconveniente de estas aproximaciones es que los resultados de la ejecución especulativa deben descartarse cuando los datos no pueden ser validados, y tener que realizar profiling del código se convierte en una carga software innecesaria.

La explotación de ILP en tiempo de ejecución se lleva a cabo en los procesadores superescalares actuales [Joh91]. En los procesadores superescalares, las instrucciones se buscan, se decodifican, se renombran, y se envían a las colas de ejecución, donde se ejecutan cuando sus operandos están disponibles. Esta manera de detectar ILP es más flexible puesto que el hardware posee información completa acerca de las dependencias entre las instrucciones (en contraposición con la información estática limitada que posee el compilador) [RF]. Si exploramos las tendencias de los procesadores superescalares actuales, veremos que existe una disparidad sobre cómo explotar ILP y mejorar el rendimiento, todo ello sin incurrir en una complejidad excesiva en el circuito que pueda llevarnos a limitar la velocidad de funcionamiento del procesador [AHKB00]. Por un lado, los procesadores Alpha 21464 [Eme99] e IBM Power4 [Kah99] apuestan por lanzar a ejecutar hasta 8 instrucciones en cada ciclo de reloj. Por otro lado, el procesador Intel Pentium4 [Int00] es un intento de explotar al máximo la frecuencia de reloj con un lanzamiento limitado de instrucciones en cada ciclo de reloj. Este procesador tiene una trace cache [PW94] que envía hasta 3 micro-operaciones por ciclo de reloj a la unidad de ejecución, pero, a cambio, consigue un tiempo de ciclo muy bajo usando supersegmentación.

Paralelismo de thread

Otra fuente de paralelismo es el paralelismo de thread (TLP). Un programa presenta TLP si puede ser descompuesto en diferentes threads, o grupos de instrucciones, que

pueden ejecutarse concurrentemente, tanto si es de forma especulativa como si no. Esta aproximación produce un sistema con una productividad alta y una mejor utilización de los recursos.

Uno de los tipos de TLP es el multithreading simultáneo (SMT) [TEL95]. En este caso, instrucciones de threads diferentes coexisten a la vez en el buffer de reordenación. En cada ciclo de ejecución, el procesador lanza a ejecutar instrucciones que pueden pertenecer a threads diferentes. Para realizar una implementación adecuada de SMT son necesarios unos pocos bits para cada instrucción, que mantengan la información de thread, así como tablas de renombre diferentes para cada uno de los threads. Un ejemplo de esta tendencia es el procesador Alpha 21464 [Eme99], según ha anunciado la compañía que lo fabrica. Este procesador Alpha puede ejecutar hasta 8 instrucciones en un ciclo de reloj. Para explotar este potencial de ejecución, este procesador utiliza ejecución de instrucciones fuera de orden y técnicas de búsqueda de instrucciones especiales para crear cuatro unidades virtuales de procesamiento de threads, lo que hace que la CPU se asemeje a un sistema multiprocesador de 4 vías. El mayor problema de esta propuesta es el soporte software que se necesita para extraer threads de las aplicaciones.

Existe también otra aproximación para explotar el paradigma TLP. Se denomina multiprocesamiento en un chip (CMP) y se basa en poner más de un procesador en un solo chip. IBM utiliza CMP en el procesador Power4 [Kah99], como también hace Compaq en el procesador Piranha [BGM+00]. Esta propuesta no posee el inconveniente del soporte software, pero incurre en el coste extra que supone poner dos procesadores en un único chip.

Paralelismo de datos

Finalmente, la tercera fuente de paralelismo en un programa es el paralelismo de datos (DLP). Un programa posee paralelismo de datos cuando existe un trozo de código (normalmente un bucle) que ejecuta la misma operación sobre una serie de datos [PH96] [Fly97]. Estos datos se encuentran normalmente en estructuras de datos tipo vector o matriz. La operación a realizar puede llevarse a cabo sobre datos consecutivos en

memoria o sobre elementos que están separados en memoria una cantidad fija de posiciones, llamada stride. El número de elementos sobre el que se realiza la operación se denomina longitud vectorial. Explotar DLP en bucles disminuye el número de instrucciones y operaciones que se han de ejecutar. Además, las instrucciones vectoriales proporcionan una gran cantidad de trabajo a las unidades funcionales del procesador (tanto como indique la longitud vectorial), lo que mantendrá las unidades funcionales ocupadas durante un gran número de ciclos.

Aunque muchos procesadores han utilizado la explotación de DLP en su diseño [Rus78] [IW91] [Oed92], solo dos compañías están dedicadas actualmente a la fabricación de procesadores que explotan paralelismo a nivel de palabra: NEC [vdSD01c] y Cray [BS00]. Sin embargo, en los últimos años han surgido las populares extensiones del repertorio de instrucciones que tratan con paralelismo intra palabra, como MMX [PW96], Altivec [NJ99], VIS [Koh95] y MDMX [MIP97]. Esta es también una forma de paralelismo de datos en la que datos de pequeño tamaños se empaquetan en un único registro del procesador y se realizan las operaciones simultáneamente sobre los distintos elementos.

Con todo, algunos de los paradigmas de paralelismo que acabamos de discutir son ortogonales entre sí, y en la practica se implementan juntos en un mismo procesador. Por ejemplo, el IBM Power4 [Kah99] posee dos procesadores en un único chip, cada uno de ellos capaz de lanzar a ejecutar varias instrucciones por ciclo. El procesador Alpha 21464 [Eme99] ejecuta hasta cuatro threads simultáneamente, y cada uno de ellos puede lanzar a ejecutar hasta ocho instrucciones fuera de orden en cada ciclo de reloj. En el caso límite, podríamos tener varios procesadores en un único chip, cada uno de ellos podría explotar multithreading simultáneo, y podría también extraer ILP de los programas lanzando a ejecutar varias instrucciones en cada ciclo de reloj. Esta es una idea importante que nos lleva a concluir que existen diversas fuentes naturales de paralelismo en los programas, y cada procesador trata de alcanzar niveles altos de rendimiento explotando algunas de estas fuentes. En esta tesis nos basamos en esa idea y proponemos el diseño de un procesador que explota paralelismo de datos junto con la ejecución superescalar tradicional. Este procesador estará acoplado a una jerarquía de memorias cache especialmente diseñada, que estará orientada a acceder a datos

escalares y vectoriales. Nuestro diseño alcanza valores de rendimiento para aplicaciones numéricas y de multimedia que un procesador superescalar no puede alcanzar por sí mismo.

Dado que nuestra propuesta está basada en la fusión del ILP más DLP, a continuación presentamos un estudio más profundo de estos dos paradigmas.

A.1.2 Paradigma ILP

El paradigma ILP se explota principalmente en las procesadores superescalares actuales [Joh91] [SS95]. Estos procesadores utilizan diferentes técnicas orientadas a la detección y explotación de paralelismo de instrucciones:

- **Segmentación o pipelining**. La ejecución segmentada de un flujo de instrucciones explota ILP, puesto que diferentes partes de diferentes instrucciones se ejecutan simultáneamente. En una ruta de datos segmentada la ejecución de cada instrucción se lleva a cabo siguiendo una serie de etapas secuenciales [Kog81] [PH96] [Fly97]. El número exacto de etapas depende del diseño de cada procesador [DF90]. Sin embargo, los pasos principales que se llevan a cabo en la ejecución de una instrucción son: búsqueda, decodificación y renombre de registros, issue, ejecución y almacenamiento de resultados. En algunos procesadores estas etapas pueden ser dividas además en etapas adicionales, dependiendo de las decisiones de diseño. En la etapa de búsqueda de la instrucción, se accede a la cache de instrucciones para leer la instrucción. La etapa de decodificación y renombre de registros realiza la decodificación de la instrucción, renombra los registros lógicos a registros físicos del procesador y predice el resultado del salto para poder tomar una decisión acerca de la ejecución de las instrucciones siguientes (si se continua la ejecución secuencialmente o si se salta a la dirección de destino del salto). Después de eso, se envía la instrucción a la colas de ejecución de las distintas unidades funcionales. Entonces la instrucción se ejecuta en la unidad funcional apropiada (o accede a la cache de datos si se trata de una instrucción de acceso a memoria). La instrucción termina

su ejecución escribiendo el resultado en el banco de registros, si es que produce algún resultado que deba ser almacenado.

- **Ejecución de múltiples instrucciones.** El modelo segmentado que hemos comentado se puede evolucionar para que sea capaz de tratar con más de una instrucción en cada ciclo de reloj [SV87]. En ese caso, se buscan, decodifican, renombran, se ejecutan y se terminan varias instrucciones en cada ciclo. Por supuesto, las instrucciones que se ejecutan en paralelo no pueden depender unas de las otras. En los primeros años de la década de los 90 se fabricaron algunos procesadores, como el DEC 21064 [McL93], el HP PA 7100 [AAD+93] y el MIPS R8000 [Hsu94], que realizaban la ejecución de múltiples instrucciones, siempre que fueran de diferente tipo. Los procesadores actuales son capaces de ejecutar múltiples instrucciones por ciclo de reloj, aunque sean del mismo tipo. La mayoría de ellos lanzan cuatro instrucciones en cada ciclo, aunque existen algunos, como el Itanium [Sha99] que puede lanzar a ejecutar hasta seis instrucciones por ciclo de reloj.

- **Planificación dinámica de instrucciones y ejecución fuera de orden.** La técnica anterior, junto con la segmentación de la ejecución de las instrucciones, permite explotar mucho más ILP, pero está fuertemente basada en la habilidad para encontrar varias instrucciones independientes que puedan ejecutarse en paralelo. Este proceso de búsqueda de instrucciones independientes puede llevarse a cabo estáticamente, en tiempo de compilación, o dinámicamente, en tiempo de ejecución. En el primer caso, el compilador selecciona grupos de instrucciones independientes y las empaqueta en una única instrucción VLIW [PSW91] [Gas91]. En el otro caso, se buscan las instrucciones, se decodifican y se envían a las colas de ejecución en el orden original del programa, pero una vez están allí, pueden ejecutarse fuera de orden siempre que sus operandos estén disponibles. Después de la ejecución, las instrucciones se gradúan en el orden original del programa de forma que se preserve la semántica del programa [DT92]. Esta técnica le proporciona al procesador una flexibilidad adicional a la hora de encontrar instrucciones independientes que puedan ejecutarse, manteniendo así las unidades funcionales ocupadas.

Todas estas técnicas combinadas permiten al procesador alcanzar altos niveles de rendimiento. Por supuesto, los procesadores podrían alcanzar aún mejor rendimiento

si los programas se compilaran conociendo las características de la máquina donde se va a realizar la ejecución. Por ejemplo, el compilador puede aplicar algunas técnicas de compilación, como desenrollado de bucles o reemplazamiento escalar, como forma de exponer una mayor cantidad de paralelismo de instrucción, lo que facilita la explotación de ILP en tiempo de ejecución.

Los dos mayores retos de un procesador que explota ILP son los saltos y los accesos a la memoria. Estos dos elementos interrumpen de algún modo el flujo de instrucciones e impiden que las instrucciones avancen por el pipeline. Los fallos de cache son un problema debido a que paran el pipeline, disminuyendo así el rendimiento. Se han desarrollado diversas técnicas para luchar contra el efecto de los fallos de cache y disminuir así el tiempo de acceso a la memoria [CB92] [SV97] [SC97] [Zha96] [Vei97] [RBS96]. Sin embargo, siempre existe un cierto número de fallos de cache que no pueden eliminarse debido a la primera vez que se accede a los datos (fallos forzosos).

En cuanto a los saltos, el problema son los fallos en la predicción del salto. La predicción de forma precisa del resultado de los saltos es un tema de investigación muy activo [McF93] [YMP93] [SJSM96] [Zha96] [JSN98], en el que hay un gran interés en la comunidad investigadora. Cada vez que se falla en la predicción de un salto, y se está ejecutando el camino equivocado, es necesario vaciar el pipeline del procesador. En esas situaciones existe una pérdida de rendimiento debido a los ciclos de ejecución empleados en ejecutar las instrucciones del camino equivocado.

A.1.3 Paradigma DLP: Otra fuente de paralelismo

El paralelismo de datos (DLP) utiliza técnicas de vectorización para descubrir paralelismo de palabra en un programa especificado secuencialmente, y expresa este paralelismo utilizando instrucciones vectoriales [Rus78] [Oed92] [CGMW88] [NKT+95] [WKI86] [AJ88] [DH] [SK86] [Smi91] [CTS96]. Una única instrucción vectorial especifica una operación que se llevará a cabo, repetidamente, sobre una serie de datos. Cada operación realizada sobre cada elemento individual es independiente del resto de operaciones y, por lo tanto, una instrucción vectorial es fácilmente segmentable y altamente paralela [Arn83] [HT72] [Ric78] [GBH96] [NKT+95].

Existen dos ventajas muy importantes en la utilización de instrucciones vectoriales para expresar paralelismo de datos. La primera es que se reduce el número total de instrucciones que tienen que ejecutarse para completar un programa, dado que cada instrucción vectorial posee más contenido semántico que las instrucciones escalares correspondientes. En segundo lugar, el que las operaciones individuales de una instrucción vectorial sean independientes entre sí permite una ejecución más eficiente: una vez que se lanza a ejecutar una instrucción vectorial en una unidad funcional, ésta estará ocupada con trabajo útil durante una gran cantidad de ciclos. Durante esos ciclos, el procesador puede ir buscando otras instrucciones vectoriales que puedan ser lanzadas a ejecución en la misma unidad funcional, o en otra. Es muy probable que, en el momento en que una instrucción vectorial termine su trabajo, exista ya otra instrucción vectorial preparada para ser ejecutada y ocupar así la misma unidad funcional. Sin embargo, en un procesador escalar, cuando una instrucción se envía a una unidad funcional, se requiere otra instrucción en el ciclo siguiente para poder así mantener las unidades funcionales ocupadas. Desafortunadamente, pueden ocurrir muchos riesgos que impidan poder encontrar estas instrucciones independientes con la rapidez con que se requiere: dependencias de datos, fallos de cache, fallos en la predicción de los saltos, etc.

La combinación de estos dos efectos tiene muchas ventajas relacionadas:

- En primer lugar, se disminuye la presión en la unidad de búsqueda de instrucciones. Especificando muchas operaciones en una única instrucción, se reduce el número total de instrucciones que es preciso ejecutar. Además, desaparecen muchos saltos, integrados en la semántica de las instrucciones vectoriales [QEV98b].
- Una segunda ventaja es la simplicidad de la unidad de control. Con relativamente poco esfuerzo de control, una arquitectura vectorial puede controlar la ejecución de muchas unidades funcionales diferentes, ya que muchas de ellas trabajan en paralelo de forma completamente síncrona.
- La tercera ventaja está relacionada con la forma de acceder a los datos en memoria: una instrucción vectorial puede especificar de forma exacta un secuencia larga de direcciones de memoria. Por consiguiente, el hardware posee conocimiento, por adelantado, sobre las referencias a memoria, y puede planificar estos accesos de

forma eficiente [VLL+92] [PVAL95] [CEV98]. Además, en el modo de acceso a memoria DLP, cada dato que el procesador pide se utiliza finalmente para realizar algún cálculo. No existe ningún tipo de prebúsqueda, como el que aparece en las líneas de la cache. Por último, la información sobre el patrón de acceso utilizado en los accesos a memoria se envía al hardware a través del stride, y éste puede usar esa información para mejorar el rendimiento del sistema de memoria [VLL+92] [VLPA95] [PVAL95].

- Otra ventaja más es que una operación vectorial de memoria puede amortizar la latencia de memoria sobre un gran conjunto de datos vectoriales. Dado que cada instrucción vectorial funciona sobre una larga serie de datos, las latencias de las unidades funcionales y de la memoria se pueden amortizar sobre todos los elementos vectoriales. En el caso particular de los accesos a memoria, una vez se comienza una operación de carga de datos de memoria, se paga una latencia inicial, pero entonces, suponiendo que no haya conflictos, se obtiene una palabra de datos en cada ciclo de reloj.
- Por último, el modelo DLP puede escalarse fácilmente a niveles mayores de paralelismo haciendo réplicas de las unidades funcionales y añadiendo caminos más anchos desde los registros vectoriales a las unidades funcionales. Todo esto puede hacerse sin tener que incrementar la complejidad o la presión en la unidad de decodificación. El contenido semántico de las instrucciones vectoriales ya incluye la noción de operaciones paralelas. Este aumento puede ser tan grande como lo sea la longitud vectorial.

Todas estas ventajas nos llevan a considerar que vale la pena incluir técnicas DLP en los microprocesadores futuros. Además, incluir técnicas DLP no impide explotar también ILP, lanzando a ejecutar varias instrucciones en cada ciclo de reloj. Nuestra propuesta consiste en mezclar técnicas ILP y DLP en un único procesador en un chip. Creemos que añadir DLP a un procesador ILP puede llevarnos a explotar mayores niveles de paralelismo. Aunque nuestra propuesta no lo incluye, adicionalmente podrían aplicarse también técnicas TLP para alcanzar aún mayores niveles de rendimiento.

A.1.4 Objetivo de la tesis

El objetivo de esta tesis es mostrar que las técnicas ILP y el DLP pueden mezclarse en una única arquitectura para ejecutar código regular vectorizable con un nivel de rendimiento que cada paradigma, por separado, no podría alcanzar. Trataremos de mostrar que la combinación de los dos tipos de técnicas produce un rendimiento alto con un coste y una complejidad reducidos: la arquitectura resultante posee una unidad de control relativamente sencilla, tolera muy bien la latencia de memoria y puede particionarse fácilmente en bloques regulares para superar así los problemas de retardo de las implementaciones VLSI futuras. Además, la simplicidad del control y la regularidad de la implementación ayudan ambas a conseguir un tiempo de ciclo reducido. Más aún, mostraremos que esta arquitectura se puede escalar muy fácilmente, mientras que escalar un procesador ILP es muy costoso en términos hardware (y, en cierto sentido, no puede realizarse). Mostraremos que, incluso si escalamos un procesador superescalar sin tener en cuenta las consideraciones anteriores, el rendimiento que se obtiene está muy por debajo del rendimiento que obtiene la máquina que explota ILP y DLP.

Para alcanzar un rendimiento alto en la arquitectura ILP+DLP que se propone, es un punto clave de esta tesis la propuesta de una nueva jerarquía de cache, especialmente diseñada para la explotación de los paradigmas ILP y DLP. Cada fuente de paralelismo utiliza una estrategia diferente en los accesos a los datos de memoria. Por lo tanto, la explotación de cada fuente de paralelismo requiere un diseño especial de la jerarquía de memoria. En nuestro caso, proponemos una jerarquía de memoria adaptada para el acceso a datos escalares y vectoriales. Esta jerarquía de memoria está basada en la cache vectorial, o vector cache, que es una cache orientada a realizar accesos a datos escalares y vectoriales. Esta cache podrá proporcionar un gran ancho de banda al banco de registros vectorial, permitirá escalar este ancho de banda a medida que escalamos las unidades funcionales, minimizará los conflictos entre los datos escalares y vectoriales y garantizará que el tiempo de ciclo del procesador no tendrá que aumentarse por el hecho de incluir un puerto de gran ancho de banda con el banco de registros vectorial.

A.1.5 Síntesis de la tesis

Esta tesis realiza una contribución en el campo de la microarquitectura de los procesadores. El objetivo es mostrar que las técnicas ILP y DLP pueden mezclarse en una única arquitectura para ejecutar aplicaciones numéricas y multimedia con un nivel de rendimiento que cada paradigma, por separado, no podría alcanzar. Proponemos un procesador superescalar de la generación actual mejorado con una unidad vectorial. La arquitectura propuesta se acopla con un nuevo diseño de una jerarquía de cache que incluye una cache vectorial.

El primer tema que desarrollamos es un estudio detallado de las características, a nivel de repertorio de instrucciones, de un conjunto de aplicaciones numéricas y de multimedia. Este estudio ha mostrado que las versiones vectoriales de los programas ejecutan menos bloques básicos, menos instrucciones y menos operaciones que las versiones escalares. Por lo tanto, nuestra propuesta de procesador necesitará unidades de búsqueda y decodificación de instrucciones menos agresivas, lo que nos puede llevar a conseguir un menor tiempo de ciclo del procesador.

El análisis dentro de las regiones S (escalares) y D (vectoriales, DLP) muestra que las versiones escalares de los programas ejecutan muchos más bloques básicos, muchas más instrucciones y muchas más operaciones dentro de las regiones D que las versiones vectoriales de los mismos programas. El análisis de las regiones S muestra que la calidad del código escalar generado por el compilador escalar es mejor que la del código generado por el compilador vectorial. La solución a este inconveniente consiste en construir una versión híbrida de los programas de prueba, a partir de las versiones vectorial y escalar puras. Cada programa vectorial híbrido consta de las regiones S originales de la versión escalar del programa, más las regiones D de la versión vectorial del programa. Los programas vectoriales híbridos ejecutan menos instrucciones y operaciones que los programas vectoriales puros, mientras siguen manteniendo las mismas características vectoriales.

A partir de estos estudios afirmamos que, dadas las ventajas que ofrece el repertorio de instrucciones vectorial, se justifica la exploración de la posibilidad de añadir una

unidad vectorial a una arquitectura superescalar actual. Por lo tanto, realizamos el diseño de un procesador superescalar con una unidad vectorial y lo comentamos.

A continuación presentamos los resultados en forma de rendimiento de nuestra propuesta de arquitectura superescalar con una unidad vectorial (SSV), comparada con un procesador superescalar tradicional (SS). Por un lado, el estudio de escalabilidad y rendimiento potencial de las arquitecturas SSV y SS, con un sistema de memoria ideal, muestra que la arquitectura SSV escala muy bien, a medida que se añaden al procesador más recursos de memoria y de cómputo. Además, alcanza unos valores de paralelismo mayores que la arquitectura SS con un coste y una complejidad menores.

El análisis de las regiones S y D también muestra que la arquitectura SSV alcanza mayores valores de paralelismo dentro de las regiones D. Sin embargo, su contribución al rendimiento global viene determinado por el peso relativo de las regiones D en el programa completo. Tal y como se esperaba, el rendimiento dentro de las regiones S es mejor para la arquitectura SS. La razón es que el núcleo superescalar de la arquitectura SSV permanece constante a lo largo de las diferentes configuraciones, mientras que en la arquitectura SS el núcleo superescalar se escala añadiendo más y más recursos.

Por otro lado, estudiamos también el rendimiento de la propuesta SSV cuando se introduce una jerarquía de cache real. Hemos estudiado dos jerarquías de cache diferentes, ambas basadas en el diseño de la cache vectorial. Los dos modelos difieren en dónde se coloca la cache vectorial, si en el primer de nivel de la jerarquía o en el segundo nivel. Como un primer paso estudiamos la eficiencia de la jerarquía de cache en ambos modelos. Este estudio muestra que los programas numéricos ejercen una mayor presión en la memoria principal, y que el modelo con la vector cache en el segundo nivel de la jerarquía reacciona mucho mejor a esta presión produciendo un menor tráfico con la memoria principal y unas tasas de acierto en la cache más altas.

El estudio del tiempo de detención de la cache vectorial revela que esta cache puede estar detenida durante un largo porcentaje del tiempo total de ejecución debido a diversas razones. Las dos razones principales son el llenado de las estructuras del MSHR y

de los buffers de escritura. Las otras dos razones (coherencia y conflictos de cache) contribuyen en menor medida a esas detenciones.

La evaluación del rendimiento con una memoria real muestra de nuevo que los programas numéricos están limitados por la memoria, mientras que los programas multimedia no lo están. El estudio dentro de las regiones S y D muestra que, para la arquitectura SSV, los programas escalan bien dentro de las regiones D y se comportan de manera constante dentro de las regiones S.

Como resultado general de estos estudios concluimos que la arquitectura SSV es una arquitectura factible desde el punto de vista del rendimiento. Alcanza un mejor rendimiento que la arquitectura SS tradicional, tanto con sistemas de memoria ideal como real, a medida que se escala la configuración del procesador. Es una buena opción para programas multimedia y alcanza muy buenos resultados para los programas numéricos.

No obstante, estos resultados pueden aún mejorarse ajustando los parámetros de la jerarquía de memoria de manera que se eliminen los cuellos de botella que han sido detectados. Aumentando los mecanismos no bloqueantes de la memoria (esto es, aumentando el número de entradas en el MSHR y en los buffers de escritura) se obtiene una disminución del efecto negativo que producía el llenado de los MSHR y los buffers de escritura. Las mejoras de rendimiento obtenidas con estos mínimos cambios alcanzan hasta un 37% para los programas que usan registros vectoriales de 128 elementos. Añadir un puerto de memoria extra para realizar accesos escalares es una mejora beneficiosa para la ejecución de programas heterogéneos (que podrían incluir incluso programas con baja vectorización) en la máquina SSV. Este puerto escalar adicional proporciona una mayor flexibilidad en los accesos a memoria. Aumentar el ancho de banda con la memoria en el procesador SSV proporciona un aumento del rendimiento en aquellos programas que presentaban un gran tráfico con la memoria principal.

Estudiamos también el efecto de las evoluciones futuras en los microprocesadores, en lo que a nivel de integración de transistores en un chip se refiere. El aumento de la integración provocará cambios en el diseño de la jerarquía de memoria, incluyendo un aumento del tamaño, de la asociatividad y de la latencia de las memorias cache; y al

mismo tiempo, un aumento en el ancho de banda con memoria. Los programas que están limitados por el ancho de banda de memoria mejoran su rendimiento. Mientras tanto, el resto de programas sufrirán una pérdida de rendimiento debido a que prevalece el aumento de la latencia sobre el menor tráfico con la memoria principal.

Finalmente, estudiamos dos jerarquías de cache adicionales para tratar el problema de los accesos a memoria vectoriales con stride distinto de la unidad. Los resultados muestran que estas jerarquías de memoria, aunque son más caras de implementar, pueden alcanzar mejoras del 3% al 18% sobre la jerarquía de memoria básica. Cuanto más se ajusta el diseño de la cache para tratar el problema de los accesos con stride, mejores son los resultados obtenidos. Aunque estas caches son complejas, y costosas de implementar, las mejoras que se obtienen a cambio hacen que valga la pena plantearse su implementación en el futuro, cuando la tecnología disponible lo permita.

A.2 PLANTEAMIENTO Y METODOLOGíA UTILIZADA

En esta sección presentamos la metodología utilizada en la investigación, así como las diferentes herramientas utilizadas. Hemos utilizado simulación alimentada por trazas como base de nuestra metodología de experimentación. Los estudios desarrollados en esta tesis han sido realizado comparando el rendimiento de las simulaciones alimentadas por trazas de las arquitecturas ILP e ILP+DLP. Cuando, en alguna sección, se propone una mejora sobre lo previamente propuesto, es interesante mostrar la mejora sobre la propuesta básica. En ese caso se incluyen también los resultados de la propuesta básica.

Herramientas de generación de trazas y de simulación

Como acabamos de mencionar, hemos utilizado simulaciones alimentadas por trazas de ejecución, como base de nuestra metodología de experimentación. Las ventajas de utilizar simulaciones alimentadas por trazas son la facilidad para reproducir los resultados y el entorno controlado en el que se realizan los experimentos.

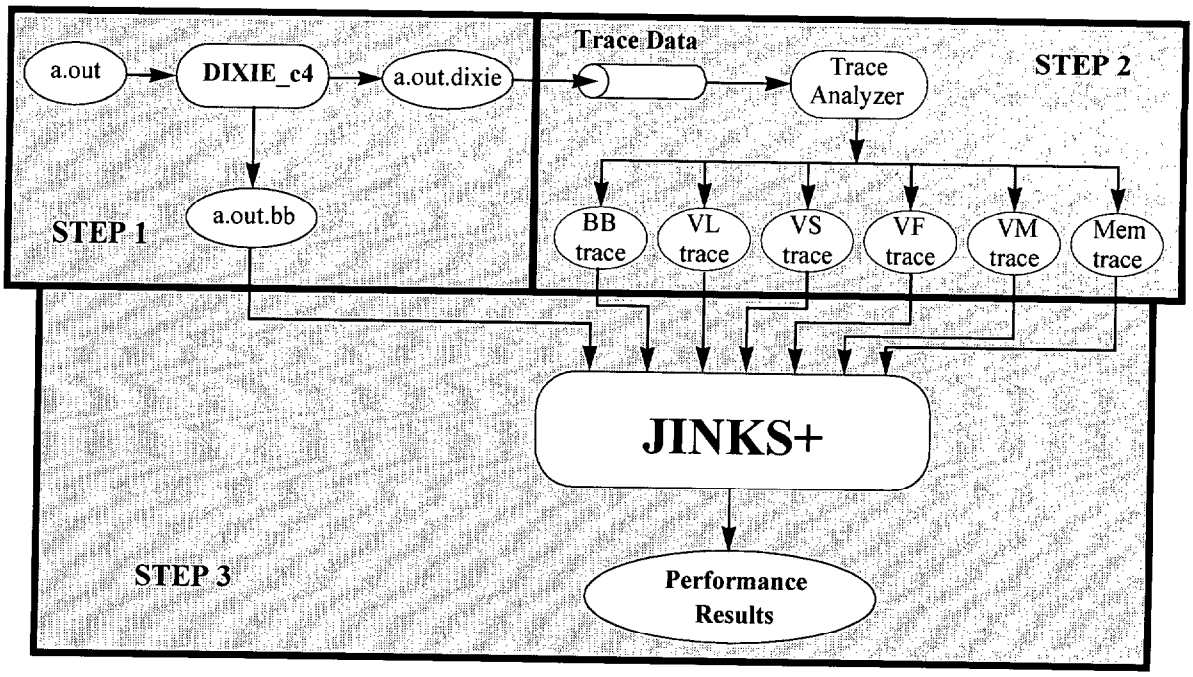


Figure A.1 El proceso de instrumentación.

En concreto, hemos utilizado dos herramientas para obtener trazas y una herramienta para realizar las simulaciones. La primera herramienta de obtención de trazas es *Dixie-c4*, que es una herramienta tipo pixie, adaptada para el Convex C4 [Con93] a partir de una versión previa [EM94]. Esta herramienta produce trazas estáticas de programas binarios que se ejecutan en modo vectorial en un único procesador Convex C4. En concreto, produce una traza de los bloques básicos ejecutados, así como una traza de los valores contenidos en el registro de longitud vectorial (VL). Las trazas obtenidas se almacenan físicamente en diferentes ficheros en memoria. La ventaja de usar trazas estáticas es que, una vez obtenidas las trazas, no es necesario disponer de una máquina Convex C4 para realizar las simulaciones. Sin embargo, la desventaja que posee este tipo de trazas es que los ficheros que contienen las trazas son demasiado grandes, si la traza es muy larga. Este hecho además se agrava si necesitamos varias versiones de las trazas de cada programa, como es nuestro caso.

El proceso para extraer las trazas y realizar la simulación se muestra en la figura A.1: los programas se compilan en el Convex C4 con optimización -O2, que permite vectorización. Después los ejecutables se procesan usando Dixie-c4, que descompone los

ejecutables en bloques básicos e instrumenta los bloques básicos para producir seis tipos de trazas: la traza de bloques básicos (BB), la trazas de los valores almacenados en los registros de longitud vectorial (VL), stride vectorial (VS), vector first (VF) y máscara vectorial (VM), y la traza de todas las referencias a memoria (realmente es una traza de las direcciones base de todas las referencias a memoria) (Mem). Dixie instrumenta todos los bloques básicos de los programas, incluyendo el código de las librerías, lo que es importante porque las llamadas a las rutinas altamente vectorizadas que ofrece el sistema operativo también se analizan y se incluyen en nuestros estudios.

Una vez los ejecutables han sido procesados con Dixie-c4, los ejecutables modificados se ejecutan en el Convex C4. Estas ejecuciones son las que producen las trazas en sí, que representan de forma precisa la ejecución de los programas.

La segunda herramienta para obtener trazas es Atom [SE94], una herramienta propia de Digital que solo puede ejecutarse en esas máquinas. Esta herramienta se ha utilizado para obtener trazas dinámicas de binarios escalares ejecutándose en un procesador Alpha 21264 [Gie97] [Kes99]. Esta herramienta proporciona un juego de herramientas diverso que permite realizar desde la cuenta de los bloques básicos que se han ejecutado hasta un modelado del comportamiento de la cache. Proporciona la infraestructura común para todas las herramientas de instrumentación de código, dejando que el usuario especifique los detalles. El usuario especifica los puntos en el programa donde desea realizar la instrumentación, las llamadas a procedimiento que se deben realizar y los argumentos que hay que pasar a esos procedimientos. A partir de estas especificaciones se generan trazas dinámicas, lo que significa que los elementos de la traza son pasados directamente del programa al simulador, sin necesidad de utilizar ficheros de disco. Aunque esta característica elimina los largos ficheros de traza de las trazas estáticas, es necesario ejecutar tanto el programa como el simulador en un computador Digital.

Por último, la tercera herramienta es un simulador general parametrizable, llamado Jinks+, desarrollado como una mejora del simulador previo Jinks [Esp95]. Este simulador incluye un núcleo superescalar, y la posibilidad de añadir o no unidades vectoriales. Posee también módulos de simulación muy precisos de la jerarquía de memoria, utilizando diversos modelos de memoria posibles.

Las trazas obtenidas, usando tanto Dixie-c4 como Atom, alimentan al simulador Jinks+, que es capaz de modelar el comportamiento de la ejecución de un programa en una arquitectura ILP y en una arquitectura ILP+DLP. Realiza una simulación, a nivel de ciclo, de ambas arquitecturas y mantiene información detallada sobre la utilización de los recursos, número de ciclos de ejecución, etc. Simulando ambas arquitecturas con la misma herramienta conseguimos eliminar los errores que podrían acumularse por el hecho de utilizar simuladores diferentes.

A.2.1 Las arquitecturas utilizadas: Convex C4 y Alpha EV6

La arquitectura vectorial Convex C4 [Con93] es una arquitectura del tipo registro-registro. Consiste en una unidad escalar y una unidad vectorial conectadas a través de un único puerto de memoria a un sistema de memoria entrelazado. En la arquitectura se definen tres tipos diferentes de registros: A, S y V. Los registros A (registros de direcciones) se usan para generar la dirección base de todas las referencias a memoria. Existen 32 registros de tipo A, de 32 bits cada uno. Los registros escalares (registros S) son de 64 bits y se utilizan para contener todos los cálculos realizados en modo escalar, tanto enteros como de coma flotante. Los registros vectoriales (registros V) contienen hasta 128 palabras, de 64 bits cada una. Estos registros se utilizan en todos los cálculos vectoriales, enteros y de coma flotante. Existen 16 registros vectoriales.

La unidad escalar ejecuta todas las instrucciones que implican registros escalares (A y S). Existe una cache de datos de 32 KB que se utiliza sólo para los accesos a datos escalares. La unidad vectorial posee tres unidades de cómputo y comparte la unidad de acceso a memoria con la unidad escalar. Una de las unidades de cómputo es una unidad aritmética de propósito general capaz de ejecutar todas las instrucciones vectoriales. Las otras dos son unidades funcionales restringidas que no ejecutan todas las instrucciones.

Las instrucciones vectoriales se realizan bajo el control del registro de longitud vectorial. Este registro escalar de 8 bits indica la longitud deseada de las instrucciones vectoriales (desde 0 hasta 128). El registro vector first también controla la ejecución de las instrucciones vectoriales puesto que indica el primer elemento de los registros

vectoriales sobre el que ejecutar la operación. Las operaciones vectoriales de memoria utilizan un tercer registro de control, el registro de stride vectorial. Este registro indica la distancia, en bytes, que separa los datos que deben buscarse en memoria. Por último, el registro de máscara vectorial es un registro de 128 bits que permite hacer todas las operaciones bajo el control de una máscara. Cada bit de este registro determina si cada operación individual realizada es válida o no, de acuerdo con una condición.

Por otro lado, el procesador Alpha 21264 [Gie97] [Kes99] sigue una arquitectura RISC de load/store de 64 bits diseñada haciendo un énfasis particular en la velocidad del reloj, el lanzamiento de múltiples instrucciones en cada ciclo, la ejecución fuera de orden y los múltiples procesadores.

Cada procesador Alpha 21264 posee un conjunto de registros que mantienen el estado actual del procesador. Existen 32 registros enteros, cada uno de 64 bits. Existen también 32 registros de coma flotante, cada uno de 64 bits. Las instrucciones son muy simples y son todas de 32 bits. La memoria se accede a través de direcciones de byte de 64 bits, utilizando la convención little-endian o big-endian. Las direcciones virtuales, tal como las ve el programa, se traducen a direcciones físicas de memoria mediante el mecanismo de manejo de memoria. La unidad básica direccionable es el byte (8 bits). Las operaciones de memoria son cargas o almacenamientos, y toda la manipulación de datos se realiza a través de los registros.

La implementación de esta arquitectura ejecuta las instrucciones fuera de orden. El procesador puede buscar, decodificar, renombrar y lanzar a ejecutar hasta cuatro instrucciones en cada ciclo de reloj. Existen cuatro unidades de ejecución enteras y cuatro de coma flotante a las que se pueden lanzar hasta cuatro instrucciones en cada ciclo. La cache de datos de primer nivel es de 64 KB y acepta cualquier combinación de dos operaciones de carga/almacenamiento en cada ciclo. En total, pueden estarse ejecutando en cualquier momento hasta 80 instrucciones a la vez, más 32 cargas, más 32 almacenamientos.

A.2.2 Conjunto de programas de prueba

Hemos elegido un conjunto de programas numéricos y de multimedia como conjunto de programas de prueba. Dentro de los programas numéricos elegimos los SPECfp92 [CKDK91] y los Perfect Club [BCK+89] . Aunque nos habría gustado utilizar programas SPEC más actuales, el problema del almacenamiento de las trazas estáticas nos impidió hacerlo. En cuanto al conjunto de programas multimedia, utilizamos los programas Mediabench [LPMS97]. En resumen, nuestro conjunto de programas de prueba está formado por:

- **Perfect Club**: *Bdna* and *Arc2d*
- **Specfp92**: *Swim256*, *Hydro2d*, *Nasa7* and *Tomcatv*
- **Mediabench**: *Jpeg Encode Jpeg Decode Epic* and *Gsm Encode*

A.2.3 Modificaciones de los programas de prueba

Hemos tenido que realizar diversos cambios a los programas de prueba originales para realizar nuestros estudios. Estos cambios han sido: algunas modificaciones en los programas multimedia para poder vectorizarlos, el stripmining manual de todos los programas para utilizar distintas longitudes vectoriales y la separación de las regiones S y D en todos los programas.

De estas modificaciones vale la pena comentar el stripmining manual, que nos permitirá cortar los bucles de forma que obliguemos al compilador a utilizar una longitud vectorial menor que la máxima longitud vectorial disponible. Esta es una manera de simular que disponemos de registros vectoriales menores de los que en realidad existen.

En cuanto a la separación de la regiones S y D, la motivación viene de que una máquina DLP realmente extrae un mejor rendimiento de las regiones D de los programas. En este sentido, nos interesa estudiar cuanto paralelismo puede alcanzarse en esas regiones, sin que se vean influenciadas por las regiones S. Esta separación de regiones se llevó a cabo mediante un proceso de varios pasos que requiere la codificación, a mano, de las

fronteras entre cada dos regiones. Cuando estos programas se ejecutan, se recogen las trazas y se realizan simulaciones, nos damos cuenta de que se produce un interesante fenómeno de solape entre cada dos regiones adyacentes. Este fenómeno debe tratarse con cuidado para realizar una medida tan fiable como se pueda.

A.2.4 La calidad del código escalar en los programas vectoriales

Dado que realizamos comparaciones de programas ejecutados con dos compiladores diferentes, uno más viejo y otro más nuevo, era de esperar que la calidad del código generado por el compilador más viejo fuese peor. Así fue efectivamente el caso. El código escalar generado por el compilador vectorial adolece de ciertas optimizaciones que el compilador escalar sí utiliza.

La solución adoptada para disponer de unos programas vectoriales de calidad fue construir programas vectoriales híbridos procedentes de una mezcla de las dos versiones. Cada programa híbrido posee como parte escalar las regiones S del programa generado con el compilador escalar, y como parte vectorial las regiones D del programa generado por el compilador vectorial.

A.2.5 La métrica de rendimiento EIPC

Comparar la ejecución de programas escalares y vectoriales usando la métrica estándar de instrucciones por ciclo (IPC) no tiene sentido debido a la diferencia semántica entre las instrucciones de ambas arquitecturas. La solución adoptada consistió en utilizar un único valor de instrucciones (el número de instrucciones de la máquina SS), variando el número de ciclos de ejecución. Esta medida tiene el significado intuitivo de "lo bien que debería funcionar una maquina superescalar para lograr llegar al rendimiento de la arquitectura ILP+DLP". De este modo, la medida EIPC se define como:

$$EIPC\ SSV = \frac{NumTotal\ instrucciones\ SS}{NumTotal\ ciclos\ SSV} \tag{A.1}$$

Para realizar el estudio dentro de las regiones S y D también es preciso definir el EIPC dentro de las regiones, que se realiza de la forma siguiente:

$$EIPC\ SSV\ regiones\text{-}D = \frac{instrucciones\ SS\ en\ regiones\text{-}D}{ciclos\ SSV\ en\ regiones\text{-}D} \tag{A.2}$$

$$EIPC\ SSV\ regiones\text{-}S = \frac{instrucciones\ SS\ en\ regiones\text{-}S}{ciclos\ SS\ en\ regiones\text{-}S} \tag{A.3}$$

A.3 PRINCIPALES APORTACIONES

Las principales aportaciones de esta tesis son un estudio en profundidad de la comparación entre repertorios de instrucciones escalar y vectorial, una propuesta de arquitectura ILP+DLP con un sistema de memoria especialmente diseñado para ella, y una evaluación de esta propuesta utilizando tanto memoria perfecta como la jerarquía de memoria que se propone.

A.3.1 Comparación de los repertorios de instrucciones escalar y vectorial

Aunque las arquitecturas se han evaluado de manera tradicional utilizando medidas de rendimiento, es interesante en este caso detenerse a realizar en primer lugar un estudio detallado que compare los repertorios de instrucciones escalar y vectorial. Este estudio nos permitirá analizar la diferente naturaleza de ambos repertorios de instrucciones. El estudio está enfocado en los siguientes temas: ventajas de los repertorios de instrucciones vectoriales, caracterización de los benchmarks, caracterización vectorial, influencia de la longitud vectorial, análisis de regiones y caracterización de los benchmarks híbridos.

Ventajas de los repertorios de instrucciones vectoriales

Las ventajas de los repertorios de instrucciones vectoriales se derivan del hecho de que las instrucciones vectoriales especifican la realización de una serie de operaciones sobre un conjunto de datos, de forma que cada operación realizada es independiente del resto de operaciones. Esto conlleva una serie de ventajas en cuanto a ancho de banda de búsqueda de instrucciones, rendimiento del sistema de memoria y control de la ruta de datos.

Ancho de banda de búsqueda de instrucciones. Dado el mayor contenido semántico de las instrucciones vectoriales, se necesita un menor número de ellas para especificar una tarea. Por lo tanto, se reduce el ancho de banda necesario para realizar la búsqueda de las instrucciones, además de reducirse también el impacto negativo de los saltos y la presión en la unidad de búsqueda de instrucciones.

Rendimiento del sistema de memoria. En los procesadores superescalares uno de problemas principales es la memoria, debido a los fallos de cache y a las altas latencias. El ancho de banda que se consume trayendo de memoria a la cache las líneas de datos muchas veces no se ve recompensada pues no todos los datos traídos serán efectivamente utilizados. En un procesador vectorial, por el contrario, todos los elementos que se traen de memoria se utilizan. Además, la información del stride que se envía a la memoria puede ser utilizada por ésta para mejorar el rendimiento en los accesos a la misma [VLL+92] [VLPA95] [PVAL95]. En el caso de la latencia de memoria, una instrucción de memoria vectorial puede amortizar la latencia entre un número grande de elementos. En cuanto al ancho de banda de memoria, una máquina vectorial puede hacer un uso más eficiente del ancho de banda pues con un solo puerto a memoria puede solicitar un gran número de elementos [CEV98].

Control de la ruta de datos. Para escalar los procesadores superescalares actuales es necesario invertir muchos transistores en la implementacin de la ventana de dispatch, los buffers de reordenación, la lógica de selección de instrucciones, etc [PJS97]. En el caso vectorial, sin embargo, sólo es necesario replicar las unidades de ejecución y poner

caminos más anchos entre el banco de registros y las unidades funcionales, sin modificar la unidad de decodificación.

Caracterización de los benchmarks: bloques básicos, instrucciones, operaciones y tamaños de datos

Este estudio nos revela que los programas escalares ejecutan más bloques básicos que los programas vectoriales, para la mayoría de los programas. Las razones son: la ejecución predicada que los programas vectoriales ejecutan y el mayor contenido semántico de las instrucciones vectoriales, lo que reduce el número de bucles ejecutados.

Tal como se esperaba, los programas vectoriales ejecutan muchas menos instrucciones que los programas escalares. El uso de las instrucciones vectoriales permite realizar un bucle en un menor número de iteraciones, lo que además implica un menor número de cálculos de direcciones y control del bucle. Por este motivo, el número de operaciones ejecutadas también es menor en los programas vectoriales. El ratio entre el número de operaciones superescalares y vectoriales está entre 1.05 y 1.25, siendo de 4.14 para el programa *Gsm Encode*.

En cuanto al tamaño de datos, los dos conjuntos de programas presentan características diferentes. Los programas numéricos realizan operaciones con memoria de tamaño 64 bits, mayormente, mientras que en los programas multimedia estas operaciones se realizan principalmente con datos de tamaño 32 bits. Además, los programas multimedia ejecutan gran cantidad de operaciones con datos de tamaño 8 bits y 16 bits, en comparación con los numéricos.

Caracterización vectorial de los benchmarks

Analizadas las distintas características vectoriales de los programas, como el porcentaje de vectorización, la distribución de longitudes vectoriales utilizadas, la distribución de strides, la ejecución bajo máscara y el uso del vector first, los resultados obtenidos muestran diversos comportamientos.

Nuestros programas tienen un porcentaje de vectorización entre alto y moderado, considerando que una vectorización alta es aquella que es mayor del 70%. Los programas más vectorizables son *Swim256*, *Hydro2d*, *Nasa7*, *Arc2d* y *Jpeg Encode*, y los moderadamente vectorizables son *Bdna*, *Jpeg Decode*, *Epic* y *Gsm Encode*.

La distribución de la longitud vectorial, cuando se varía la longitud vectorial máxima, muestra patrones de comportamiento diferentes para los distintos programas. Los programas *Swim256*, *Tomcatv*, *Bdna*, *Arc2d* y *Jpeg Decode* utilizan longitudes vectoriales muy cercanas a 128. A medida que se disminuye la longitud vectorial máxima, la contribución porcentual de las longitudes vectoriales pequeñas disminuye. El resto de programas presentan una distribución más desigual. *Hydro2d* presenta una única longitud vectorial (102) que corresponde al número de punto de la malla usada en la dirección Z del problema que resuelve. *Nasa7* y *Gsm Encode* presentan una distribución en escalera; en *Jpeg Encode* la longitud vectorial dominante es 8; y *Epic* presenta una única longitud vectorial 16. Esto sugiere que, incluso entre programas vectorizables, la utilización de la longitud vectorial varía bastante.

La distribución de stride vectorial muestra que algunos benchmarks, como *Swim256*, *Hydro2d*, *Tomcatv* y *Gsm Encode*, ejecutan la mayoría de los accesos a memoria con stride 1, beneficiándose así del ancho de banda de memoria. Otros programas ejecutan accesos a memoria con stride 1 y algún otro stride pequeño, como en *Jpeg Decode* y *Bdna*. Por último en *Jpeg Encode*, *Nasa7*, *Arc2d* y *Epic* existe una gran cantidad de accesos a memoria realizados con strides grandes, y por tanto estos programas no utilizan bien el ancho de banda de memoria disponible.

La ejecución con el vector first muestra que esta capacidad sólo la utilizan los programas *Swim256*, *Hydro2d*, *Tomcatv* y *Epic*, lo que quiere decir que sólo estos programas hacen reuso de los datos de los registros vectoriales en recurrencias.

Por último, la ejecución bajo máscara muestra que en nuestros programas esta facilidad se utiliza relativamente poco. El uso más intensivo lo realiza el programa *Jpeg Decode*, que realiza más del 22% de sus operaciones bajo máscara. Los programas *Jpeg Encode*

e *Hydro2d* ejecutan 15.81% y 14.11%. El resto, o no lo usan, o el porcentaje de uso es muy pequeño.

Influencia de la longitud vectorial

Este estudio analiza la repercusión de la longitud vectorial máxima en el número de instrucciones, operaciones y en el tráfico con la memoria. El estudio muestra que, a medida que disminuye la longitud vectorial máxima, el número de instrucciones ejecutadas aumenta, aunque no llega nunca a los valores alcanzados por los programas escalares. El número de operaciones ejecutadas también aumenta al disminuir la longitud vectorial. En algunos programas, como en *Swim256*, *Hydro2d*, *Nasa7* y *Arc2d*, este incremento es importante, mientras que en el resto no lo es tanto.

También hemos estudiado el tráfico generado entre el procesador y la memoria, cuando se disminuye la longitud vectorial. Los resultados muestran que, aunque existe variación entre el número de operaciones de memoria ejecutadas, en siete de los programas el aumento del tráfico con memoria es despreciable. Dos de las excepciones son los programas *Hydro2d* y *Tomcatv*. La comparación con el tráfico generado por los programas escalares muestra que, en general, los programas escalares mueven muchos más datos con la memoria que los programas vectoriales. La tercera excepción es el programa *Jpeg Decode*, que es el único programa que genera menos tráfico con memoria en su versión escalar.

Análisis de regiones

Como ya se ha discutido, hemos separado las regiones escalares y vectoriales de los programas para poder realizar un estudio más detallado de su comportamiento. Por supuesto, la parte más interesante de este estudio será la que se refiera a las regiones vectoriales puesto que las regiones escalares no varían su comportamiento al pasar de una arquitectura ILP a una arquitectura ILP+DLP con el mismo núcleo superescalar.

Analizando las características generales de las regiones S y D hemos visto que el porcentaje de operaciones ejecutadas dentro de las regiones D es normalmente mayor que

el porcentaje de vectorización, lo cual es normal puesto que las regiones D contienen también las instrucciones escalares necesarias para realizar los cálculos escalares dentro de los bucles.

El análisis de bloques básicos ejecutados dentro de las regiones S y D muestra que en las regiones S los programas vectoriales ejecutan muchos más bloques básicos debido a la mala calidad del código escalar generado por el compilador vectorial. En las regiones D, sin embargo, los programas vectoriales ejecutan menos bloques básicos que los escalares. Los mismos comportamientos se aprecian para el recuento de instrucciones y operaciones. Además, la mayoría de las operaciones ejecutadas se encuentran dentro de las regiones D, y en el caso de los programas vectoriales estas operaciones corresponden a instrucciones vectoriales, lo que era de esperar dado el grado de vectorización de los programas.

Caracterización de los programas híbridos

Este estudio realiza la caracterización de los programas híbridos en cuanto a instrucciones, operaciones y características vectoriales. Cada programa híbrido se construye como la mezcla de las regiones S del programa escalar original más las regiones D del programa vectorial original. Estos programas híbridos son una aproximación bastante buena a lo que serían los programas reales compilados en un compilador moderno con capacidades vectoriales.

Los programas híbridos poseen un menor número de instrucciones ejecutadas que los programas vectoriales originales, lo cual es debido al menor número de instrucciones de las regiones S. La excepción son los programas *Arc2d*, *Jpeg Decode* y *Gsm Encode*, que tienen un número de instrucciones mayor. Ello es debido a que las regiones S de los programas vectoriales generaban un menor número de instrucciones ya que estos programas realizan conversiones de datos que el repertorio de instrucciones vectorial expresa utilizando un menor número de instrucciones.

Los programas híbridos ejecutan, en general, menos operaciones que los programas vectoriales originales. Las excepciones son, de nuevo, los programas *Jpeg Decode* y *Gsm Encode*, que ejecutan más operaciones por la razón anteriormente comentada.

En cuanto a las características vectoriales, estos programas híbridos apenas modifican su comportamiento, puesto que lo que los diferencia de los programas vectoriales puros son las regiones S, y las características vectoriales se extraen de las regiones D. Las diferencias que se aprecian en el porcentaje de vectorización se derivan de que el número total de operaciones que el programa ejecuta ha disminuido un poco, lo que eleva ligeramente el porcentaje de vectorización. El resto de características vectoriales no varía.

A.3.2 El diseño de la arquitectura ILP+DLP

Para describir el diseño de la arquitectura ILP+DLP hemos de centrarnos en varios aspectos. Por un lado la ruta de datos, los pipelines, el banco de registros vectorial, y por otro la jerarquía de memoria.

La ruta de datos de la arquitectura ILP+DLP

La descripción de la arquitectura ILP+DLP comienza con la descripción de la ruta de datos. La figura A.2 muestra los componentes principales de la ruta de datos para la arquitectura propuesta. Esencialmente, la arquitectura está modelada basándonos en el diseño del procesador ILP Mips R10000 [Yag96], al que le hemos añadido un banco de registros vectoriales y conectado a las unidades enteras y a las unidades de coma flotante.

El funcionamiento general de pipeline es el siguiente: las instrucciones se buscan y se envían en orden a la etapa de decodificación, donde son renombrados los registros. Cuando una instrucción entra en la etapa de decodificación se asigna una entrada en el buffer de reordenación. Las instrucciones entran y salen del buffer de reordenación en orden. La etapa de renombre consiste en traducir cada registro virtual a un registro físico usando una tabla de mapeo. Existen cuatro tablas de mapeo independientes,

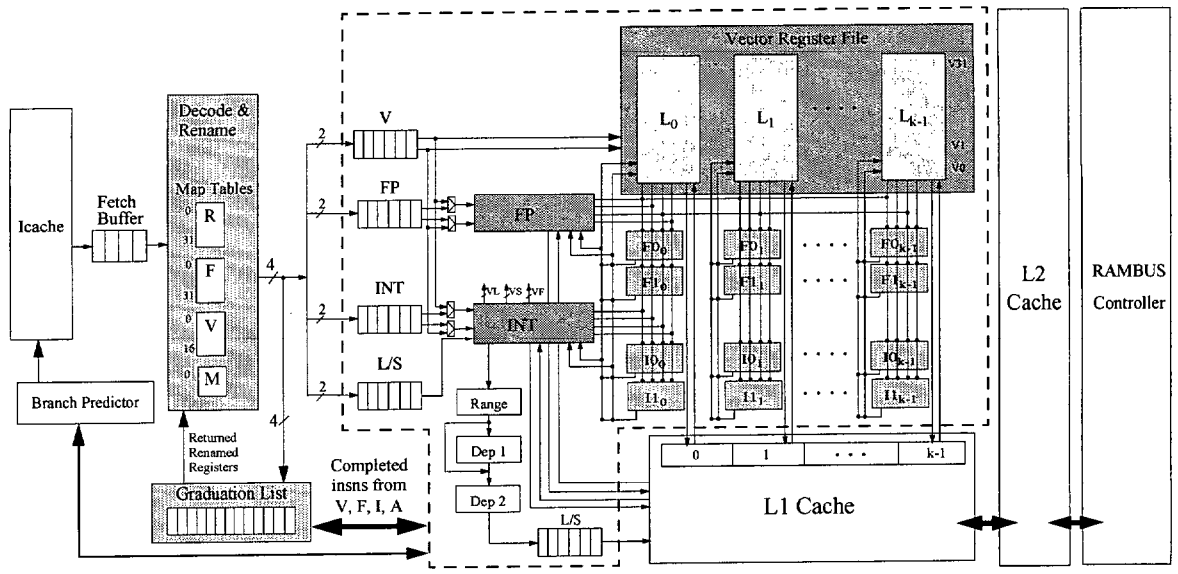


Figure A.2 La arquitectura modelizada.

una para registros enteros, una para registros de coma flotante, una para registros vectoriales y una para registros de máscara. Cada tabla mantiene su propia lista de registros libres. Cuando una instrucción define un nuevo valor de un registro lógico, la entrada en la tabla de mapeo para ese registro lógico debe ser actualizada con el nuevo registro físico. El registro físico se obtiene de la lista de registros libres apropiada y se actualiza la tabla de mapeo con ese número de registro. Además, el valor antiguo en la entrada de la tabla de mapeo se almacena en la entrada del buffer de reordenación de esa instrucción. El registro físico antiguo se devuelve a la lista de registros libres cuando la instrucción se gradúa.

Cuando una instrucción de salto entra en la etapa de decodificación y renombre, el procesador predice el resultado del salto y ejecuta especulativamente el salto basándose en la predicción realizada. El predictor de saltos es un predictor gshare [McF93] implementado de manera similar a como se implementa en la herramienta SimpleScalar [BA97].

Después de la etapa de renombre, las instrucciones se envían a una de las cuatro colas de ejecución, basándose en el tipo de instrucción. Las instrucciones esperan en estas colas

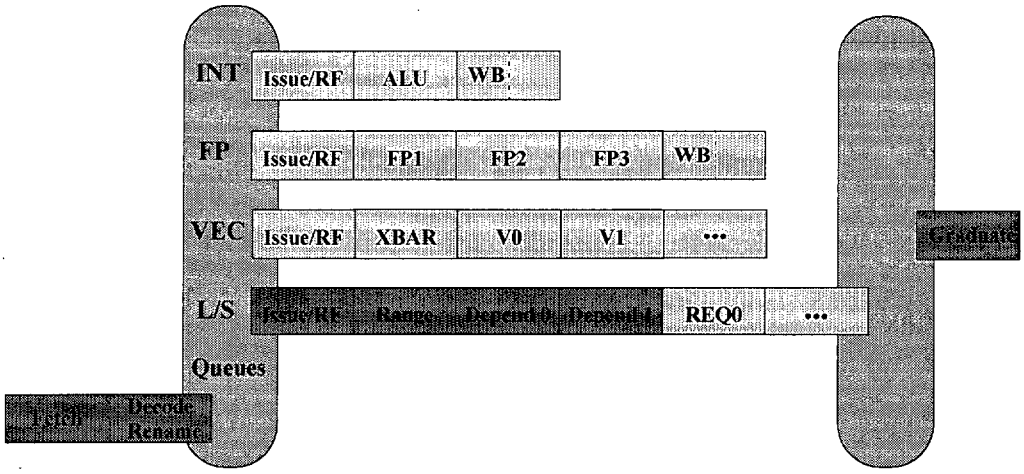


Figure A.3 Los cuatro tipos de pipelines de la arquitectura propuesta. Las etapas que se realizan en orden están marcadas en color gris oscuro.

hasta que sus operandos están disponibles y entonces pasan a ejecutarse a una de las unidades funcionales. El procesador mantiene las instrucciones decodificadas en cuatro colas de instrucciones, que lanzan dinámicamente instrucciones a ejecutar a las unidades de ejecución. Las colas permiten al procesador buscar y decodificar instrucciones a la máxima velocidad sin necesidad de pararse debido a conflictos o dependencias entre instrucciones. Las instrucciones de cada cola pueden ejecutarse fuera de orden. El procesador lanza dinámicamente una instrucción a ejecucin tan pronto como la unidad funcional esté lista, la instrucción tenga sus operandos disponibles y no dependa de otra instrucción que aún no ha terminado de ejecutarse.

En general, las instrucciones se buscan, se decodifican, se envían a las colas de ejecución y se gradúan en el orden original del programa, pero pueden ejecutarse y completarse fuera de orden.

Existen cuatro tipos de pipelines en la arquitectura propuesta, una para cada tipo de instrucción, como puede observarse en la figura A.3. Estos cuatro pipelines vienen determinados por las cuatro colas de ejecución existentes: entera, de coma flotante, de memoria y vectorial. De todas éstas, las más interesantes son la vectorial y la de memoria. La cola vectorial (al igual que las colas entera y de coma flotante) chequean el estado de todas las instrucciones en las entradas de la cola hasta que están listas

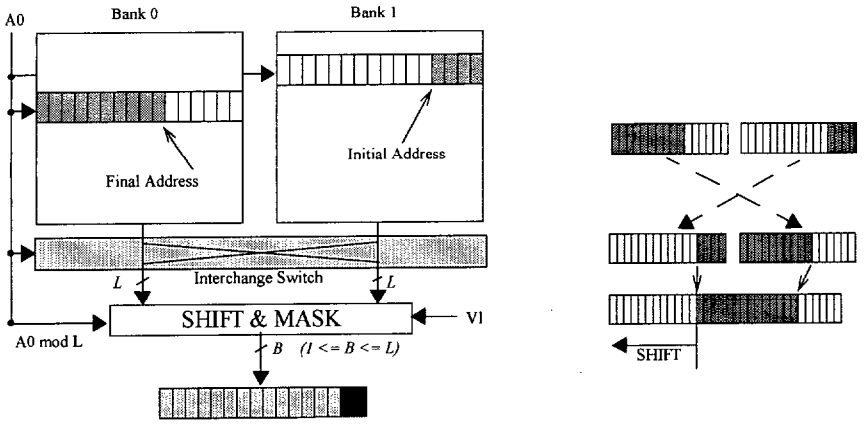


Figure A.4 La estructura de doble banco de la cache vectorial.

para ejecutarse. En ese momento se lanza a ejecutar en la unidad funcional apropiada. La cola de memoria funciona de una forma un poco diferente puesto que todas las instrucciones de memoria han de pasar por varias etapas.

El elemento principal que se ha añadido al procesador superescalar es el banco de registros vectorial, junto con sus conexiones a las unidades funcionales, y un conjunto de registros de propósito especial: el registro de longitud vectorial, el registro de stride vectorial, el registro vector first y el registro de máscara. A la hora de introducir el banco de registros vectorial es preciso elegir tres parámetros: el número de registros lógicos, el número de registros físicos y la longitud de cada registro. El número de registros lógico lo fijamos a 16, por ser un valor razonable. Algunos estudios previos realizados [EVS97] muestran que el número de registros físicos debe ser al menos el doble del número de registros lógicos, así que colocamos 32 registros físicos. En cuanto a la longitud de cada registro, es deseable que sean cuánto más largos mejor; sin embargo, dadas las restricciones de área, restringiremos nuestros estudios a registros de hasta 128 elementos.

La jerarquía de memoria propuesta

La jerarquía de memoria que proponemos es una jerarquía de memorias cache basadas en la inclusión de un nuevo diseño de cache denominado cache vectorial. La cache

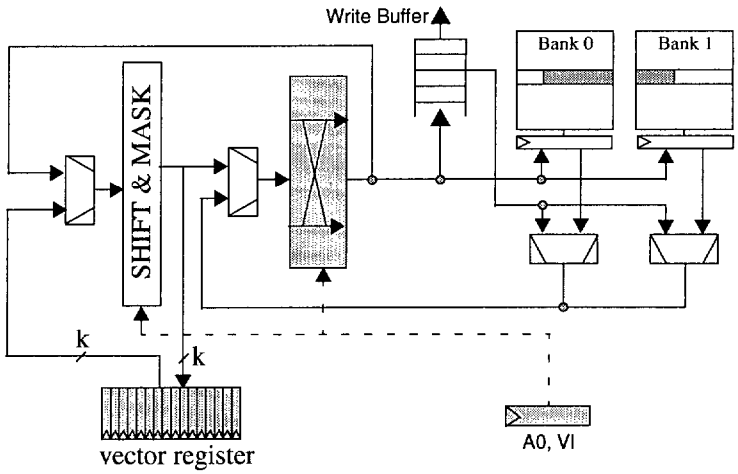


Figure A.5 Caminos de load y store de la cache vectorial.

vectorial es una cache que proporciona un gran ancho de banda, pensada para accesos a vectores con stride 1, accediendo a líneas completas de la cache en lugar de acceder de forma individual a cada uno de los elementos. A continuación, un bloque lógico de dezplazamiento y máscara alinea correctamente los datos, eliminando los residuos. La cache vectorial permite alcanzar grandes ratios de ancho de banda para accesos con stride 1, incluso para datos no alineados. Los dos puntos más importantes del diseño son la lógica de alineamiento y el impacto en el rendimiento de los accesos con stride distinto de 1.

La cache (ver figura A.4) es de doble banco, entrelazada, de forma que se puede acceder a dos líneas consecutivas a la vez. De este modo, un acceso vectorial que tenga los datos solapados en dos líneas de cache consecutivas se pueden acceder simultáneamente. Este esquema requiere tres partes: una lógica de intercambio, que se utiliza cuando necesitamos intercambiar la posición de las dos líneas, una lógica de desplazamiento, para alinear los datos accedidos a la dirección inicial que se especificó, y una lógica de enmascaramiento, que elimina los datos que no se utilizan basándose en la información suministrada por la longitud vectorial (ver figura A.5).

La vector cache posee además buffers de escritura y estructuras MSHR para manejar de forma efectiva los fallos de cache sin necesidad de bloquearse mientras resuelve estos fallos.

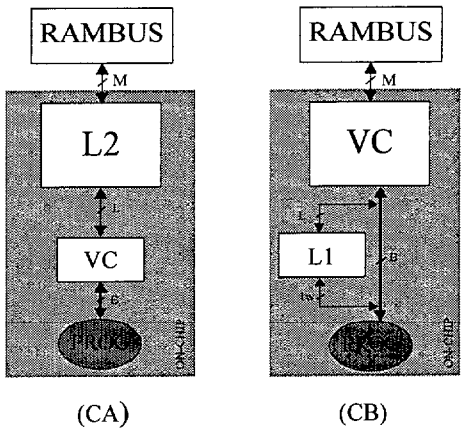


Figure A.6 La ruta de datos de la cache vectorial.

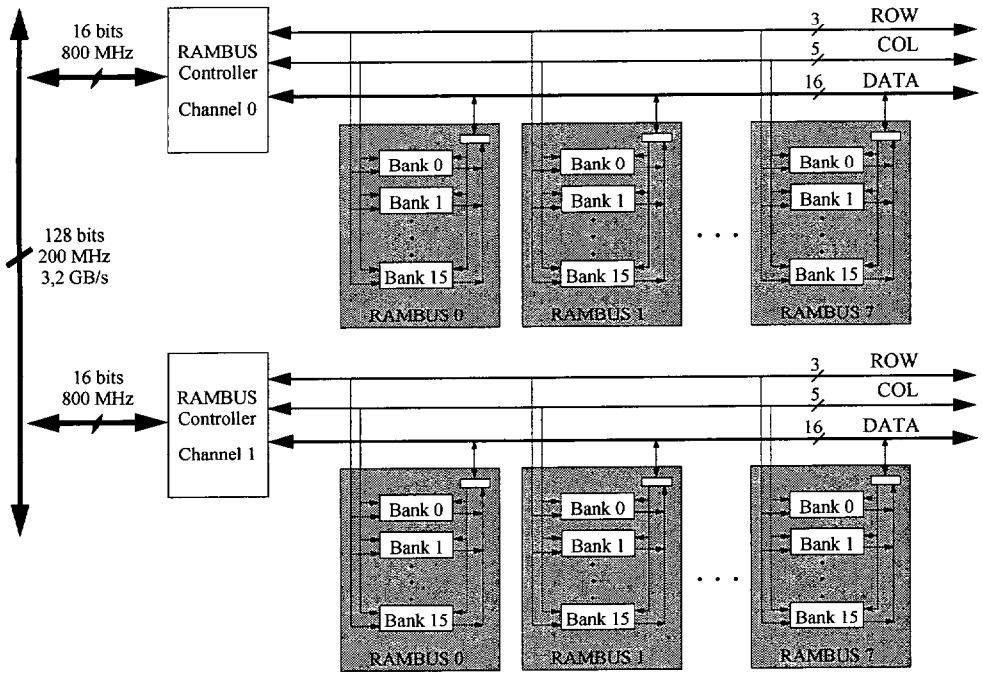


Figure A.7 El sistema de memoria principal modelizado usando tecnología RDRAM.

Esta vector cache que proponemos vamos a integrarla en una jerarquía de caches. En concreto proponemos dos jerarquías de cache, el modelo CA y el modelo CB (ver figura A.6). En el modelo CA la vector cache actúa como una cache de primer nivel, y proporciona un camino ancho entre la cache y los registros vectoriales. Tanto los

accesos escalares como vectoriales se realizan a la vector cache. Un acceso vectorial con stride 1 que acierta en la cache utiliza completamente el ancho de banda del bus y tarda en ejecutarse $\lceil VL/B \rceil$ ciclos. Los accesos escalares, o los accesos con stride distinto de 1, no pueden aprovechar todo el ancho de banda y cargan un máximo de una palabra por ciclo. La cache de segundo nivel, que se asume que está también dentro del chip, es una cache convencional que se conecta a un bus bidireccional hacia un controlador de RAMBUS [Cri97] externo (ver figura A.7). Además, obligamos a que se cumpla la propiedad de inclusión entre las caches de primer y segundo nivel.

Esta jerarquía de caches tiene las siguientes desventajas: la complejidad de la cache vectorial puede obligar a aumentar el tiempo de ciclo del procesador, puede ser difícil extender el diseño de la vector cache a una versión de múltiples puertos, realizar un almacenamiento en la cache vectorial tarda el doble de ciclos que realizar un acceso de lectura y el conjunto de datos de los programas será seguramente mucho mayor que el tamaño de la cache de primer nivel.

Por estos motivos, proponemos también otra jerarquía de caches denominada modelo CB. Este modelo posee una cache convencional en el primer nivel, y la cache vectorial en el segundo nivel. Los accesos escalares se realizan en la cache de primer nivel a razón de un dato por ciclo. Los accesos vectoriales, en cambio, se realizan en la cache de segundo nivel, haciendo un bypass de la cache de primer nivel. Estos accesos se realizan a razón de B elementos por ciclo (siendo B el ancho del bus que una la cache de segundo nivel con el procesador) si el stride es 1. Si el stride no es 1, entonces se accede a un elemento en cada ciclo. De este modo se desacoplan los datos escalares de los vectoriales, puesto que los escalares están en la cache de nivel uno, mientras que los datos vectoriales se encuentran en la cache de segundo nivel. El problema al hacer bypass es mantener la coherencia entre las dos caches, lo que se lleva a cabo realizando invalidación de las líneas de la cache de primer nivel siempre que sea necesario.

A.3.3 Rendimiento potencial y escalabilidad de la arquitectura propuesta

Una vez descrita la arquitectura y la jerarquía de memoria propuestas, pasamos a realizar su evaluación. Esta evaluación se lleva a cabo realizando en primer lugar un estudio de rendimiento y escalabilidad cuando se dispone de un sistema de memoria ideal. A continuación se realiza una evaluación con una jerarquía de memoria real, que es la descrita anteriormente.

Estudio de rendimiento y escalabilidad con un sistema de memoria ideal

El estudio de rendimiento y escalabilidad analiza cómo se comporta un diseño ILP+DLP (o SSV) a medida que se aumentan los recursos de computación y de memoria. Este estudio se lleva a cabo sobre un conjunto configuraciones que se mueven desde las configuraciones más modestas (1x2) a las más agresivas (16x32). El primer número indica el numero de palabras que pueden leerse de memoria en un ciclo y el segundo número indica el número de resultados en coma flotante que se pueden generar, teniendo en cuenta que en la máquina SSV tenemos únicamente dos unidades funcionales y lo que hacemos es replicarlas para obtener así configuraciones más agresivas en cómputo, pero no más complejas de fabricar.

Los resultados de este estudio muestran que, para la arquitectura SSV, todos los programas mejoran su rendimiento a medida que se añaden recursos al procesador. Esto es especialmente cierto para los resultados que incluyen registros vectoriales de 128 elementos. Los resultados para registros de 16 elementos, sin ser tan buenos, alcanzan también unos niveles de rendimiento bastante aceptables con muy poca inversión en área del chip (sólo 4KB para el banco de registros vectoriales).

Observamos también que los programas numéricos escalan mucho mejor que los programas multimedia. En general, los cuatro programas multimedia alcanzan niveles de rendimiento modestos, cuando se comparan con los programas numéricos. La razón principal es el porcentaje de vectorización, que según ya estudiamos, es menor para los

programas multimedia que para los programas numéricos (menos del 70%, con valores del 40% en dos de los programas). Mientras tanto, los programas numéricos alcanzan porcentajes de vectorización bastante altos (por encima del 75%, llegando en algunos casos a valores por encima del 95%). Dado que los programas multimedia son menos vectorizables, cuando se escala la arquitectura SSV existe un menor porcentaje de instrucciones que pueden beneficiarse de estos recursos adicionales, y el aumento de rendimiento global es pequeño. Además, dado que el núcleo superescalar de la arquitectura SSV permanece constante a lo largo de todas las configuraciones, la ejecución de las partes no vectorizables de los programas no contribuye a mejorar el rendimiento a medida que se escalan las configuraciones. En resumen, los programas multimedia están caracterizados por unos porcentajes de vectorización relativamente bajos, y por lo tanto, a medida que escalamos el sistema, la parte no vectorizable del código se convierte en un cuello de botella debido a la restricción en el número de instrucciones que se pueden lanzar a ejecutar en cada ciclo.

Además, ese estudio también muestra que existe una pérdida de rendimiento cuando pasamos de registros de 128 elementos a registros de 16 elementos. Esto se debe al aumento en el número total de instrucciones y operaciones que hay que ejecutar cuando se disminuye la longitud de los registros vectoriales. Esta pérdida de rendimiento es sustancial en los programas numéricos. En los programas multimedia, sin embargo, esta pérdida no es tan significativa. El caso extremo es el programa *Epic*, que no cambia su comportamiento al pasar de registros de 128 elementos a registros de 16 elementos puesto que la única longitud vectorial que se utiliza durante todo el programa es 16.

Observemos que manteniendo constante el ancho del puerto de memoria, y aumentando el número de vías de las unidades funcionales, se obtiene un aumento importante en el rendimiento. Por otro lado, manteniendo constante el número de vías de las unidades funcionales y aumentando el ancho del puerto de memoria observamos que el aumento es muy pequeño en comparación con el otro caso. La razón es que, dado que estamos estudiando un sistema de memoria perfecta, ésta puede traer de memoria la cantidad de datos necesarios en muy pocos ciclos de reloj. Dado que no se aumenta el número de vías de la unidades funcionales, no podemos procesar el número de elementos, cada

vez mayor, que la memoria envía al procesador, así que no podemos beneficiarnos del hecho de recibir cada vez más y más elementos en cada ciclo.

La comparación con la arquitectura superescalar (SS) muestra que nuestra propuesta alcanza valores de rendimiento similares para la configuración más modesta (1x2) en casi todos los programas. Sin embargo, a medida que aumentamos el número de unidades funcionales (o el número de vías en el caso SSV), el rendimiento de la arquitectura SS se aplana. Mientras tanto, nuestra propuesta mejora a una velocidad mucho mayor, con excepción de los programas *Nasa7*, *Jpeg Decode* y *Gsm Encode*. Aunque la arquitectura SS tiene más capacidad de búsqueda, decodificación, ejecución y graduación, no es capaz de disminuir el número de ciclos de ejecución que vendrá determinado, seguramente, por las dependencias entre las instrucciones del programa. En contraste, en la arquitectura SSV la ejecución de una instrucción vectorial tarda muchos ciclos. Por lo tanto, la unidad funcional está ocupada durante gran cantidad de ciclos, lo que le da tiempo suficiente al procesador para encontrar una instrucción independiente que pueda lanzar a la misma unidad funcional, tan pronto como termine la ejecución de la instrucción que está realizando. En la arquitectura SS, sin embargo, la instrucción tarda solo uno o dos ciclos en ejecutarse, lo que no da tiempo suficiente en muchos casos para encontrar otra instrucción que esté lista para ejecutarse. Esto hará, probablemente, que la unidad funcional se detenga durante un cierto número de ciclos.

Podemos decir, por lo tanto, que la escalabilidad de la arquitectura SSV es mucho mejor que la de la arquitectura SS, y además esta escalabilidad se alcanza usando una menor complejidad de la unidad de control. Con un núcleo superescalar que lanza a ejecutar 4 instrucciones en cada ciclo, fuera de orden, podemos alimentar hasta 32 unidades de coma flotante, a la vez que alcanzamos una fracción bastante grande del rendimiento pico.

El estudio de rendimiento y escalabilidad dentro de las regiones S y D también ha sido realizado, y muestra que dentro de las regiones D todos los programas alcanzan todavía un mayor rendimiento que al medir los programas completos. Algunos programas, como *Swim256*, apenas presentan diferencias cuando se comparan con el rendimiento global, puesto que son 99% vectorizables, así que el efecto de las regiones S es despreciable.

Sin embargo, otros programas, como *Tomcatv*, *Gsm Encode*, *Jpeg Decode*, *Epic* y *Jpeg Encode*, muestran un aumento de rendimiento considerable. La mayoría de estos programas posee un porcentaje de vectorización moderado, así que es normal que al añadir los efectos de las regiones S y D el rendimiento global obtenido disminuya. El resto de los programas apenas modifica su comportamiento con respecto al mostrado en el rendimiento global.

En cuanto al comportamiento dentro de las regiones S, los resultados muestran que el comportamiento de los programas para la arquitectura SSV permanece constante a medida que aumentamos la configuración. Esto se debe a que la arquitectura SSV tiene un núcleo superescalar fijo en todas las configuraciones, variándose el ancho del puerto a memoria y el número de réplicas de las unidades funcionales. Por lo tanto, la parte escalar se ejecuta siempre en las mismas condiciones. Los resultados de la arquitectura SS, sin embargo, mejoran a medida que se aumenta la configuración, lo cual se debe a una mejora del núcleo superescalar a medida que cambiamos de configuración. La comparación de la arquitectura SS con la arquitectura SSV muestra que para la configuración más baja ambas arquitecturas obtienen un rendimiento parecido. Sin embargo, al mejorar las configuraciones, el comportamiento de la arquitectura SSV permanece constante y el de la arquitectura SS aumenta, superando el rendimiento de la arquitectura SSV.

Por último, analizamos también el paralelismo de datos alcanzado dentro de las regiones vectoriales, utilizando la medida "operaciones ejecutadas por ciclo de reloj" (OPC). Esta medida nos permite estudiar el paralelismo real que cada arquitectura es capaz de alcanzar. Los resultados muestran que, a medida que se aumenta la configuración, tanto la arquitectura SSV como la SS explotan cada vez más paralelismo. Además, la arquitectura SSV explota mucho más paralelismo que la arquitectura SS, y la diferencia entre utilizar registros vectoriales de 128 elementos o de 16 elementos apenas se percibe para muchos de los programas.

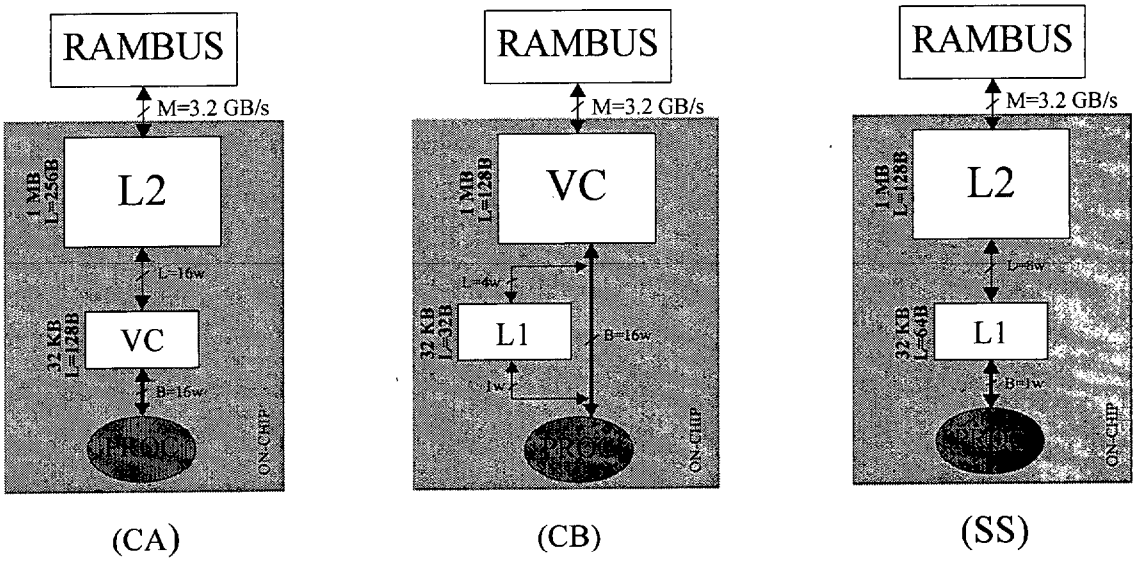


Figure A.8 Las jerarquías de memoria estudiadas para los modelos CA, CB y para la arquitectura superescalar (SS).

Estudio de escalabilidad con un sistema de memoria real

El siguiente estudio realizado es el estudio de escalabilidad conectando un sistema de memoria real al procesador ILP+DLP. Este sistema de memoria contiene dos modelos diferentes, CA y CB, que ya han sido discutidos previamente (ver figura A.8). Además, realizaremos la comparación con la arquitectura SS a la que se le ha conectado un sistema de memoria convencional consistente en dos niveles de cache. En los tres sistemas de memoria se han respetado las mismas características, que son:

- Se han utilizado los mismos tamaños para las caches que pertenecen a un mismo nivel, en los distintos sistemas de memoria.
- Se han ajustado los tamaños de líneas de cache a las distintas necesidades de cada sistema de memoria, y de cada nivel.
- Se ha utilizado la misma asociatividad en las diferentes caches de un mismo nivel, en los diferentes sistemas de memoria.

- Se han puesto las mismas latencias para las caches de primer y segundo nivel, en los distintos sistemas de memoria, asumiendo además que ambas caches están dentro del chip. Esto último no es demasiado realista para las configuraciones más agresivas, especialmente en la arquitectura SS.
- El número de ciclos que se tarda en realizar una operación de lectura y escritura se ha puesto de acuerdo al tipo de cache. La cache vectorial tarda dos ciclos en hacer una escritura y un ciclo en hacer una lectura.
- Se ha utilizado el mismo número de entradas en las estructuras MSHR y buffers de escritura en todos los sistemas de memoria.

El estudio con memoria real comienza con un estudio de eficiencia de la cache, en términos de tráfico generado con la memoria principal, porcentaje de aciertos/fallos en la cache vectorial y porcentaje de tiempo que la cache vectorial está parada.

Las medidas de tráfico con la memoria principal nos dan una idea sobre si las caches son capaces de filtrar parte del tráfico que el procesador genera. Estas medidas muestran que los programas *Swim256*, *Nasa7* y *Arc2d* generan más tráfico con la memoria que el que genera el procesador. Aunque esto pueda parecer extraño, sucede cuando existe mucha polución en las líneas de cache que se traen de memoria, no se utilizan todos los datos que se han traído de memoria principal y entonces se están trayendo en realidad más datos de los que en realidad se necesitan. También se produce debido a los conflictos en la cache.

En general, el modelo CB genera menos tráfico con la memoria que el modelo CA. El motivo es el menor número de líneas de datos que existen en la cache de segundo nivel en el modelo CA, lo que aumenta el número de conflictos en la cache de segundo nivel, y por tanto aumenta también el tráfico con la memoria principal. Los resultados muestran además que, en términos de tráfico, es mejor utilizar registros vectoriales de 16 elementos con el modelo CB, puesto que así existe mayor probabilidad de reutilizar los datos, disminuyendo así el tráfico con memoria.

Los programas multimedia se caracterizan por el pequeño número de palabras que mueven con la memoria. Estos programas no tienen conjuntos de datos muy grandes, así que la mayoría de los datos caben en la cache.

La idea importante del estudio de tráfico es que, sin importar qué modelo de memoria se utilice, CA o CB, la jerarquía de cache permite filtrar significativamente el trafico que el procesador genera, haciendo que el tráfico que se mueve con la memoria disminuya. Esto disminuye la presión ejercida sobre el sistema de memoria.

El estudio de tasa de aciertos de la cache vectorial muestra que la tasa de aciertos es mayor en el modelo CB, lo cual era de esperar puesto en el modelo CB todos los accesos vectoriales los sirve la cache de segundo nivel de 1MB de tamaño, y no como en el modelo CA, en que los sirve la cache de nivel uno de 32 KB. Sin embargo, la cache de segundo nivel se encuentra a 4 ciclos de latencia del procesador, y no a 1 ciclo como la cache de primer nivel, así que la pregunta es si esta mejor tasa de aciertos provocará un aumento final en el rendimiento. Creemos que sí, porque el código vectorial es tolerante a la latencia de memoria.

En cuanto al uso de registros vectoriales largos o cortos, se observa que usar registros vectoriales de 16 elementos proporciona una mejor tasa de aciertos en ambos modelos de memoria.

Para terminar el estudio de eficiencia de la cache, hemos analizado también la cantidad de tiempo que la cache vectorial está parada, y la razón de que lo esté. Existen diversas razones para que la cache vectorial esté parada: que se haya llenado el MSHR, que se haya llenado el buffer de escritura, que se esté resolviendo un problema de coherencia con la cache de primer nivel o que se esté resolviendo un conflicto.

En los programas multimedia se observa que la cache vectorial apenas está parada durante su ejecución. Estos resultados eran de esperar ya que hemos visto que estos programas poseen un tráfico con memoria bajo y una alta tasa de aciertos. Por el contrario, en los programas numéricos la cache vectorial está parada durante un porcentaje significativo del tiempo de ejecución. En general, el modelo de memoria CA presenta

un mayor porcentaje de tiempo de parada, con la excepción de los programas *Swim256*, *Hydro2d*, *Tomcatv* y *Bdna*. Además, observamos también que utilizar registros vectoriales de 16 elementos hace que la cache vectorial esté parada más tiempo.

En general, los programas pasan la mayor parte del tiempo parados debido a que se ha llenado el MSHR o los buffers de escritura. Los otros dos motivos apenas influyen en el tiempo total que la cache vectorial está parada. En total, la cache vectorial puede estar parada hasta un 60% del tiempo de ejecución, lo que influye en el tiempo de ejecución de los distintos programas y en el rendimiento que se obtiene.

Por último, estudiamos el rendimiento que ofrece la arquitectura ILP+DLP cuando se conecta con el sistema de memoria real que proponemos. Este estudio muestra que, en general, la arquitectura SSV (tanto para el modelo CA como el CB), obtiene un mejor rendimiento que la arquitectura SS.

En cuanto a la comparación de los modelos de memoria de la arquitectura SSV, para los programas numéricos el modelo CB obtiene mayor rendimiento que el modelo CA. El uso de registros vectoriales de 128 elementos proporciona también mejor rendimiento.Por otro lado, para los programas multimedia, aunque el modelo CA obtiene mejor rendimiento que el modelo CB, la diferencia de rendimientos es muy pequeña. El modelo CA funciona un poco mejor debido al gran número de operaciones de reducción que se realizan en estos programas.

El estudio de rendimiento dentro de regiones D arroja unos resultados muy similares a los de los programas globales, lo que significa que el rendimiento global está determinado, principalmente, por el comportamiento en las regiones vectoriales de los programas. También en este caso se obtiene mejor rendimiento para las aplicaciones numéricas cuando se utiliza el modelo CB de memoria, mientras que las aplicaciones multimedia obtienen mejor rendimiento con el modelo CA. Además, utilizar registros vectoriales de 128 elementos proporciona un mejor rendimiento, lo que era de esperar ya que en ese caso se ejecutan menos instrucciones y operaciones. En cuanto a la comparación con la arquitectura SS, ésta obtiene menor rendimiento que la arquitectura SSV tanto si se usa el modelo CA como el CB.

En cuanto al estudio dentro de las regiones S, los resultados son similares a los obtenidos con un sistema de memoria ideal. En la configuración más modesta los resultados de ambas arquitecturas casi coinciden, pero a medida que crecen las configuraciones la arquitectura SS obtiene mejores resultados, ya que el núcleo superescalar es escalado. En contraste, en la arquitectura SSV el núcleo superescalar se mantiene en todas las configuraciones, y su rendimiento permanece constante.

Por último, el estudio de rendimiento con un sistema de memoria real termina evaluando la cantidad de paralelismo real que cada arquitectura es capaz de extraer dentro de las regiones D, utilizando para ello la medida "operaciones ejecutadas por ciclo" (OPC). Lo que observamos en este estudio es que añadir un sistema de memoria real a las arquitecturas SSV y SS hace que disminuya de manera significativa la cantidad de paralelismo de datos que las arquitecturas son capaces de explotar. Esto último es especialmente cierto para la arquitectura SS, que experimenta mayores pérdidas de rendimiento que la arquitectura SSV.

A.3.4 Mejoras adicionales en el diseño del sistema de memoria

Estudiamos en las secciones anteriores que la cache vectorial podía pasar hasta el 60% del tiempo de ejecución parada, debido a varias razones, lo que ocasiona un cuello de botella en la ejecución de los programas. Trataremos cada una de las razones que provocan que la vector cache se tenga que parar. Asimismo, trataremos el efecto en el rendimiento de las tendencias futuras en materia de integración de circuitos, y estudiaremos por último unos sistemas de memoria cuyo funcionamiento hace que los programas con accesos a memoria con strides distintos de la unidad mejoren su rendimiento.

Aumento de los mecanismos no bloqueantes: MSHR y buffers de escritura

Una de las causas de parada de la cache, según hemos visto, es que las estructuras MSHR y los buffers de escritura se llenan, lo que provoca que la cache no pueda aceptar

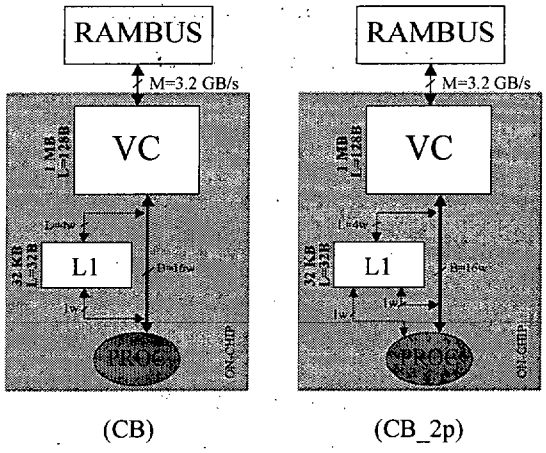
más accesos. En general, los programas numéricos son los más afectados por este tipo de cuello de botella, debido a que su conjunto de datos es bastante mayor que el de los programas multimedia.

Para reducir estos períodos de espera del procesador, debido a que la cache vectorial no acepta más peticiones de memoria, la solución más directa es aumentar el número de entradas en las estructuras MSHR y buffers de escritura. De este modo, logramos que estas estructuras tarden más en llenarse (o no se llenen), evitando así que el procesador tenga que quedar parado esperando para realizar peticiones a memoria. Los resultados que obtenemos muestran que, en general, todos los programas numéricos mejoran su rendimiento al incrementar los mecanismos no bloqueantes. Los mayores ratios de mejora aparecen para los programas *Swim256*, *Tomcatv* y *Arc2d*, que son los programas en los que el procesador pasaba un mayor porcentaje de tiempo parado debido a la cache vectorial.

El rendimiento dentro de las regiones D, al realizar esta mejora en el diseño de la jerarquía de memoria, muestra que los programas multimedia apenas se ven afectados. Sin embargo, los programas numéricos aumentan su rendimiento a medida que se aumentan los mecanismos de no bloqueo de la cache vectorial. En general, el aumento que se produce es mayor para registros vectoriales de 128 elementos que para registros vectoriales de 64 elementos. En este caso, moverse hacia registros vectoriales más pequeños implica una pérdida de rendimiento debido al mayor número de instrucciones y operaciones que deben ejecutarse.

Un puerto de memoria adicional para accesos escalares

Dado que existe una cantidad representativa de código escalar en las regiones S, especialmente en los programas multimedia, y existe también una cantidad importante de código escalar dentro de las regiones D, una forma de mejorar la ejecución de este código, que puede estar limitado por la memoria, es añadir un puerto de memoria escalar, adicional, que permita que los accesos a memoria escalares se realicen utilizando este puerto especial (ver figura A.9). De este modo estamos aumentando el número de referencias a memoria que pueden realizarse en cada ciclo: ahora puede realizarse



(CB) (CB_2p)

Figure A.9 Los modelos de memoria CB y CB_2p. El modelo CB_2p es un modelo CB mejorado con un puerto escalar adicional.

un acceso escalar por el puerto escalar, y un acceso vectorial por el puerto común escalar/vectorial.

La evaluación del rendimiento de esta propuesta muestra que las mejoras más significativas de rendimiento aparecen en los programas multimedia, concretamente en los programas *Jpeg Decode* y *Jpeg Encode*, con mejoras del 5% y 7%, respectivamente. La razón es que estos programas poseen un gran porcentaje de código escalar, de forma que pueden mejorar su rendimiento por el hecho de poseer un puerto de memoria dedicado para accesos escalares. De este modo aumentan el paralelismo en el acceso a datos. El puerto de memoria extra se utiliza tanto dentro de las regiones S como dentro de las regiones D, en paralelo con el puerto vectorial.

El efecto que provoca en el rendimiento dentro de las regiones D el haber añadido un puerto escalar extra también lo hemos estudiado. Los resultados muestran que los programas numéricos apenas mejoran su rendimiento. En los programas *Nasa7* y *Bdna* se producen mejoras de rendimiento muy ligeras. Sin embargo, los programas multimedia, como *Jpeg Decode*, *Jpeg Encode* y *Gsm Encode*, mejoran su rendimiento en un 6%, 8% y 5% respectivamente, lo que era de esperar ya que sus regiones vectoriales están polucionadas con código escalar. El puerto escalar extra proporciona flexibili-

dad adicional en el acceso a la cache de datos de primer nivel para estos programas. Aunque podríamos esperar que *Epic* se comportase del mismo modo, los problemas de coherencia de este programa impiden que se realicen esas ganancias de rendimiento que el puerto escalar extra podría aportar.

A.3.5 Aumento del ancho de banda de memoria

Identificamos previamente que existían algunos programas caracterizados porque generaban una gran cantidad de tráfico con la memoria principal. Estos programas eran principalmente los programas numéricos, y el tráfico que solicitan de memoria será servido con un ancho de banda máximo igual al que la memoria principal puede sostener. Por lo tanto, aumentar el ancho de banda con la memoria mejorará la ejecución de estos programas puesto que los accesos a memoria que les limitan será servidos con un ancho de banda mayor. Además de aumentar el ancho de banda, hemos de asegurarnos de que este ancho de banda se utiliza completamente. Para ello aumentamos también, de nuevo, el tamaño de las estructuras MSHR y buffers de escritura.

El análisis de los resultados nos muestra que, tal como se esperaba, los programas multimedia apenas se ven afectados por el incremento en el ancho de banda de la memoria. El poco tráfico que generan con la memoria ya se sirve a una tasa de transferencia elevada, así que no se benefician de un aumento en el ancho de banda de memoria.

Por el contrario, los programas numéricos mejoran su rendimiento cuando se dobla el ancho de banda con la memoria. Las mejoras más importantes aparecen en los programas *Swim256*, *Nasa7*, *Arc2d* y *Tomcatv*, con mejoras del 20.2%, 15.5%, 9% y 7.8%, respectivamente, para registros vectoriales de 128 elementos, y con mejoras del 23%, 12%, 11% y 10% para registros vectoriales de 64 elementos. De estos resultados observamos que los registros vectoriales de 64 elementos obtienen mejor rendimiento que los registros vectoriales de 128 elementos, para los programas numéricos, mientras que para los programas multimedia su rendimiento es muy similar. También observamos que un ancho de banda mayor beneficia especialmente a los programas que, o bien tienen conjuntos de datos grandes, como es el caso de los programas *Swim256* y *Tomcatv*, o

bien poseen accesos a memoria con stride distinto de la unidad, como en los programas *Nasa7* y *Arc2d*.

Una vez realizadas estas mejoras en el sistema de memoria, observamos que el tiempo de parada de la cache vectorial se reduce considerablemente, hasta el punto de que la única razón que hace que el procesador se pare debido a la cache vectorial sea los problemas de coherencia entre las caches de primer y segundo nivel. El resto de razones de parada se han reducido por debajo del 1% del tiempo total de ejecución.

Para los programas superescalares también se ha realizado el aumento del ancho de banda, y los únicos programas afectados por esta mejora son *Swim256*, *Nasa7*, *Bdna* y *Arc2d*, con mejoras del 9%, 9.5%, 7% y 21%, respectivamente. Aún así, el rendimiento que se obtiene para la arquitectura SS están muy lejos de los resultados que se obtienen con la arquitectura SSV.

El análisis de rendimiento dentro de las regiones D muestra un comportamiento similar. Los programas numéricos se benefician más de la mejora realizada en el ancho de banda de memoria, con porcentajes de mejora comparables a los obtenidos para los programas completos.

A.3.6 Análisis de los efectos de la integración en los microprocesadores

Aunque la generación actual de procesadores superescalares poseen típicamente una gran cache de segundo nivel integrada en el chip, como es el caso del procesador Alpha 21364 [Ban98] que posee una cache de segundo nivel de 1.75 MB, los avances en la lógica de integración están permitiendo introducir cada vez más transistores en un único chip. Por lo tanto, en los microprocesadores futuros podrá dedicarse gran parte del área del chip a la cache de segundo nivel. Ese es el caso, por ejemplo, del procesador Alpha 21464 [Eme99], que incluye 250 millones de transistores y que implementará una cache de segundo nivel aún mayor que la del 21364. Estas caches tan grandes se implementarán utilizando múltiples bancos de memoria, lo que favorece un aumento

natural de la asociatividad de la cache, pero que, desafortunadamente, implica también un aumento de la latencia.

Las mejoras futuras proporcionarán, además, un mayor ancho de banda. Siguiendo con el ejemplo del procesador Alpha 21464 [Eme99], sus 1100 pines permitirán a la memoria transferir datos al procesador con un ancho de banda de 12 GB/s, como mínimo. Otro ejemplo es el IBM Power4 [Die99], que con su cache de tercer nivel fuera del chip, y un puerto a esta cache de 16 bytes de ancho, también proporciona más de 10 GB/s de ancho de banda con la memoria.

Hemos realizado el estudio de nuestra cache vectorial básica de 1 MB de tamaño, y hemos doblado y cuadruplicado este tamaño de cache, a la vez que incrementamos las latencias correspondientes, para analizar qué efecto tendrá en el rendimiento el uso de estas caches en el futuro.

Los resultados muestran que el efecto de aumentar la cache de segundo nivel, así como el ancho de banda de memoria, benefician especialmente a aquellos programas limitados por la memoria. Los programas *Swim256*, *Nasa7*, *Tomcatv*, *Bdna* y *Arc2d* mejoran su rendimiento en un 13%, 13%, 15%, 3% y 5%, respectivamente, para la cache de 2MB, y usando registros vectoriales de 128 elementos. El uso de registros vectoriales de 64 bits también mejora el rendimiento, pero en porcentajes un poco menores (desde un 2.2% hasta un 10.5%).

Para los programas que no están limitados por memoria, sin embargo, no compensa el mayor número de ciclos de latencia de la cache de segundo nivel con el mayor tamaño o el mayor ancho de banda de memoria. Estos programas ya se ejecutaban bien porque cabían en la cache de 1MB, así que aumentarla no les ha beneficiado, pero sí les ha perjudicado el aumento experimentado en la latencia de la cache de segundo nivel. Las pérdidas de rendimiento para estos programas son bastante bajas, siendo del orden de 1.8%, 3.2%, 3% y 0.1% para los programas *Hydro2d*, *Jpeg Encode*, *Epic* y *Jpeg Decode*, independientemente de la longitud de los registros vectoriales.

Al aumentar el tamaño de la cache de segundo nivel hasta 4 MB el comportamiento se repite. Los programas limitados por la memoria, como *Swim256*, *Nasa7*, *Tomcatv*, *Bdna* y *Arc2d* aumentan su rendimiento un 12.4%, 12.5%, 11.1%, 2.9% y 4.5%, respectivamente, para registros vectoriales de longitud 128. Para registros vectoriales de 64 elementos los aumentos de rendimiento van desde el 2.1% hasta el 10.7%.

La comparación de los resultados para caches de 2 MB y 4 MB muestran que los resultados de la cache de 2 MB son un poco mejores que los de 4 MB. Aunque la latencia de la cache ha experimentado un aumento del 50% (de 8 ciclos a 12 ciclos), los resultados de rendimiento apenas se ven afectados. Por lo tanto, hemos alcanzado un punto en la configuración que, con una latencia larga, y con el mismo tiempo de ciclo del procesador, hace que no valga la pena gastar más área del chip en aumentar el tamaño de la cache de segundo nivel.

Los resultados dentro de las regiones D son similares a los obtenidos para los programas completos. Tal como ocurría para el rendimiento global, los programas que están limitados por la jerarquía de memoria mejoran su rendimiento a medida que se mejoran los parámetros de la cache de segundo nivel. El resto de los programas sufre una pérdida de rendimiento debido a la influencia de las altas latencias.

A.3.7 Sistemas de memoria para accesos con stride no unitario: Caches secundarias Collapsing y Multi-Address

Mientras que en las secciones previas hemos solucionado los problemas de memoria usando una aproximación más directa, aumentando el ancho de banda con la memoria, doblando el número de entradas en las estructuras MSHR y buffers de escritura, y usando caches más grandes, en esta sección vamos a tratar de hacerlo de una manera más inteligente. Teniendo en cuenta que hemos visto que una buena parte del tráfico con la memoria se debe a accesos a memoria vectoriales con stride distinto de la unidad, y que nuestra cache vectorial (que se diseñó para que fuese sencilla) no se comporta muy bien con este tipo de accesos vectoriales, en esta sección evaluamos dos jerarquías

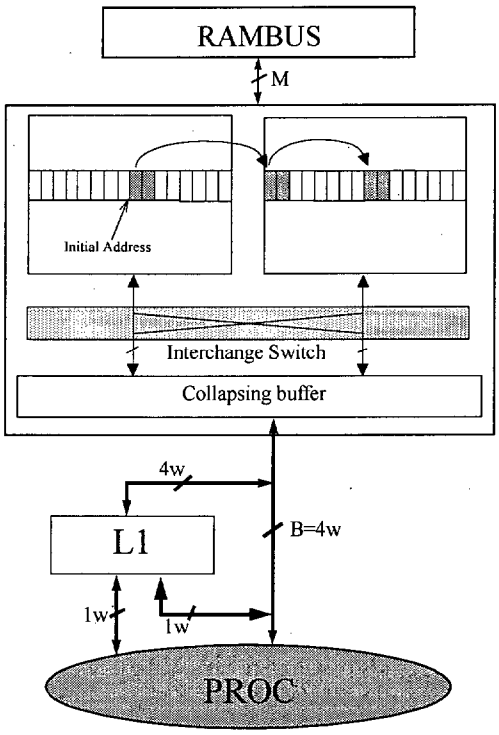


Figure A.10 La ruta de datos de la cache vectorial collapsing (CVC).

de cache alternativas, que están pensadas para mejorar el rendimiento cuando existen accesos a memoria vectoriales con stride distinto de la unidad.

Es interesante comentar que estos diseños de memorias cache son más complejos que la vector cache que hemos propuesto. Sin embargo, hemos visto también que a medida que la tecnología evoluciona, será posible integrar más transistores en un único chip, lo que permitirá integrar funcionalidad adicional a las caches. Estos diseños de cache que proponemos podrían ser parte de la funcionalidad que se añada a las futuras memorias cache.

La primera alternativa a la cache vectorial (ver figura A.10) es la Cache Vectorial Collapsing (CVC) [CEV99]. Utiliza un buffer collapsing, propuesto por Conte [CMMP95], que permite recuperar varios elementos vectoriales que se encuentran en dos líneas de cache consecutivas, incluso aunque estos elementos vectoriales no sean consecutivos. En lugar de la lógica de desplazamiento y máscara que posee la cache vectorial, el buffer

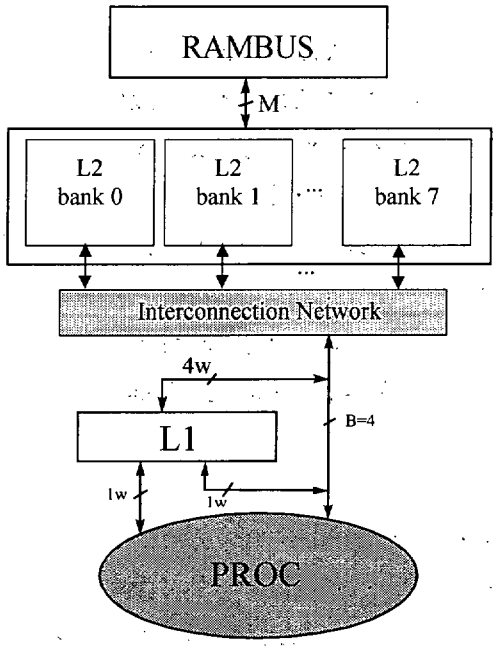


Figure A.11 La ruta de datos de la cache Multi-Address (MAC).

collapsing agrupa juntos los elementos pedidos. Este diseño será útil para accesos con strides entre 2 y 2×tamaño de línea-1, dado que en estos casos existirá más de un elemento de los solicitados en dos líneas consecutivas de la cache. Para valores de stride iguales o mayores al doble del tamaño de la línea de cache, el buffer collapsing no añade ninguna mejora a la cache vectorial básica que ya se ha evaluado.

La segunda alternativa (ver figura A.11) es la Cache MultiAddress (MAC) [CEV99]. Se trata de una cache convencional en la que un acceso a memoria vectorial de desacopla entre los diferentes puertos disponibles. Este modelo se beneficia de todos los recursos disponibles, dado que podemos enviar direcciones de memoria independientes sin tener en cuenta el stride que exista entre ellas. Esta alternativa es, por tanto, más flexible que la CVC ya que el acceso a los diferentes elementos solicitados no depende del stride que exista entre ellos. Por supuesto, este diseño es aún más complejo y caro que la cache CVC, pero mostrará la ganancia potencial de rendimiento que se puede obtener con un sistema de memoria apropiado.

La evaluación del rendimiento obtenido con estos dos modelos (CVC y MAC) muestra que ambos diseños mejoran el rendimiento. Además, CVC aumenta el rendimiento entre un 5% y un 90.6% para registros vectoriales de 128 elementos. Las mayores mejoras de rendimiento se producen en los programas multimedia, *Jpeg Decode*, *Gsm Encode*, *Jpeg Encode* y *Epic*, con mejoras del 90.6%, 75.7%, 46.1% y 21%, respectivamente. Los programas numéricos alcanzan menores mejoras de rendimiento, siendo estas mejoras del 18.2%, 17.5%, 10.4% y 9.2% para los programas *Swim256*, *Nasa7*, *Arc2d* y *Tomcatv*, respectivamente. Con registros vectoriales de 64 elementos se alcanzan mejoras más discretas, entre el 4.2% y el 90.4%.

La cache Multi-Address (MAC) mejora más el rendimiento que la CVC, alcanzando mejoras en el rango (16.4%, 128%) para registros vectoriales de 128 elementos. Los programas *Arc2d*, *Swim256*, *Jpeg Decode*, *Gsm Encode* y *Nasa7* alcanzan las mayores mejoras de rendimiento, 128%, 127.5%, 90.7%, 79.8% y 67.6%, respectivamente, sobre la cache vectorial básica. Estos programas se benefician de la flexibilidad de la cache MAC que no está restringida a un cierto rango de valores para el stride, sino que, en general, favorece los accesos con cualquier stride. Para registros vectoriales de 64 elementos las mejoras de rendimiento son algo más bajas, pero aún alcanzan valores en el rango (5.8%,109%).

El análisis en las regiones D muestra que, a medida que proporcionamos mayor flexibilidad en el número y tipo de accesos a memoria que se pueden realizar, el rendimiento aumenta paralelamente. Comparando con los resultados obtenidos para programas completos, observamos que aunque se obtiene un comportamiento muy similar, los programas *Tomcatv*, *Gsm Encode*, *Jpeg Decode*, *Epic* y *Jpeg Encode* alcanzan valores mayores de rendimiento.

La cache CVC mejora el rendimiento en casi todos los programas, desde un 0.5% hasta un 45.8% para registros vectoriales de 128 y 64 elementos. Las mejoras se concentran en los programas que poseen stride 2, 3 y 4, que son los que se benefician del hardware collapsing.

La cache Multi-Address mejora el rendimiento para todos los programas. Las mejoras mayores se producen en programas que, o bien tienen strides muy grandes (*Bdna* un 23.6%, *Nasa7* un 67.6% y *Arc2d* un 129.5%), o están fuertemente limitados por la memoria (*Swim256* un 128% y *Tomcatv* un 76.6%).

A.4 CONCLUSIONES Y TRABAJO FUTURO

La escalabilidad de los procesadores superescalares es costosa y fuertemente dependiente de la tecnología. Las arquitecturas superescalares actuales no pueden evolucionarse simplemente aumentando el número de instrucciones que se ejecutan en cada ciclo de reloj. Las tendencias actuales en el diseño de los procesadores muestran que se están explorando fuentes alternativas de paralelismo que están disponibles en los programas. Estas tendencias identifican el uso incipiente de paralelismo de instrucciones (ILP) junto con otras formas de paralelismo, como el multithreading simultáneo o multiprocesadores en un chip.

Basándonos en este análisis hemos presentado el paradigma del paralelismo de datos (DLP) como una alternativa que merece la pena explorar, de forma que la propuesta consiste en estudiar una máquina que utilice a la vez ILP y DLP. El DLP tiene ciertas ventajas inherentes, como la disminución del número de instrucciones y operaciones ejecutadas, una menor presión en la unidad de búsqueda de instrucciones, una unidad de control más sencilla, un conocimiento por adelantado de los datos de memoria que se van a acceder, un uso del 100% de los datos que se traen de memoria, la capacidad de amortizar largas latencias sobre un gran conjunto de datos y la facilidad de escalar el diseño.

La contribución principal de esta tesis consiste en demostrar que ILP y DLP pueden unirse en una única arquitectura para ejecutar aplicaciones numéricas y multimedia con un buen nivel de rendimiento.

La comparación de los repertorios de instrucciones escalar y vectorial ha mostrado que los programas vectoriales ejecutan menos bloques básicos, instrucciones y operaciones

que los programas escalares debido al mayor nivel semántico de las instrucciones vectoriales. Estos estudios iniciales del repertorio de instrucciones nos llevan a concluir que, dadas las ventajas de los repertorios de instrucciones vectoriales, vale la pena explorar la posibilidad de incluir una unidad funcional vectorial en una arquitectura superescalar actual.

Otra contribución de esta tesis es la identificación, separación y estudio de las regiones S y D para cada programa. La posibilidad de identificar y estudiar separadamente el comportamiento de un programa dentro de las regiones S y D nos ha permitido comprender y predecir mejor su respuesta en rendimiento.

El diseño de la arquitectura ILP+DLP es muy similar al de un procesador superescalar actual. La principal diferencia viene del banco de registros vectoriales añadido, así como sus conexiones con las unidades funcionales presentes en la arquitectura. También hemos añadido algunos registros de propósito especial: el de longitud vectorial, el de stride, el vector first y el registro de máscara.

Esta arquitectura ILP+DLP la hemos conectado a un sistema de memoria basado en un nuevo diseño de cache, llamada cache vectorial, que es capaz de enviar pequeños vectores al procesador a través de un bus ancho. La cache vectorial es una memoria de doble banco de la que se leen líneas completas de datos. Estas líneas se intercambian, se desplazan y se les pasa una máscara, según sea necesario, y se envían entonces al procesador. De esta forma se consigue un enlace con la memoria con un ancho de banda alto y una latencia baja.

Hemos presentado los resultados de rendimiento de nuestra arquitectura ILP+DLP y la hemos comparado con un procesador superescalar tradicional, estudiando la escalabilidad y el rendimiento potencial que puede obtenerse cuando se utiliza un sistema de memoria ideal. Los resultados muestran la arquitectura ILP+DLP escala muy bien a medida que se añaden más memoria y más unidades funcionales, sobre todo dentro de las regiones D. Además, obtiene mayores valores de paralelismo que la arquitectura superescalar a la vez que limita el coste y la complejidad del diseño.

Cuando añadimos un sistema de memoria real al diseño ILP+DLP el comportamiento sigue siendo mejor que en el procesador superescalar, para los dos modelos de memoria propuestos. En ambos modelos se ha visto que la jerarquía de cache es capaz de filtrar el trafico del procesador a la memoria. Los programas multimedia poseen un tráfico con la memoria principal más bajo que los programas numéricos, aunque en los programas numéricos la cantidad de tráfico se reduce al utilizar el modelo de memoria CB. Los estudios de acierto/fallo de cache también muestran que los programas multimedia aciertan con más frecuencia en la cache que los programas numricos, y que el modelo CB presenta mejores resultados. En general, los programas numéricos ejercen una mayor presión en la memoria principal, y el modelo CB reacciona mucho mejor a esta presión con niveles de tráfico menores y tasas de acierto más altas.

El estudio de la cache vectorial muestra que esta cache puede estar hasta un 60% del tiempo de ejecución parada debido a distintas razones, de las cuales las dos principales son que las estructuras MSHR y los buffers de escritura se llenan.

El estudio de rendimiento corrobora que los programas numéricos están limitados por la memoria, mientras que los programas multimedia no lo están tanto. Los valores de rendimiento obtenidos siguen siendo más altos que los valores obtenidos por el procesador superescalar tradicional.

El modelo de memoria CB presenta un mejor rendimiento para los programas numéricos que el modelo CA, mientras que para los programas multimedia las diferencias entre ambos modelos son muy pequeñas. Por ello proponemos el modelo CB como el sistema de memoria que se debería incluir en nuestra propuesta ILP+DLP.

Los resultados de rendimiento pueden mejorarse aún más realizando ciertas mejoras en el sistema de memoria. Estas mejoras van desde aumentar el tamaño de las estructuras MSHR y los buffers de escritura, incluir un puerto de memoria adicional para accesos escalares, aumentar el ancho de banda o incluir ciertas modificaciones a la vector cache para que realice los accesos vectoriales con stride mayor que uno de una manera más eficiente. Todas estas mejoras aumentan el rendimiento que se obtiene de la arquitectura ILP+DLP.

Por todo ello, concluimos que la arquitectura ILP+DLP es una arquitectura factible desde el punto de vista del rendimiento. Alcanza un mejor rendimiento que una arquitectura superescalar tradicional, tanto con memoria ideal como real, a medida que se escala la configuración del procesador. Es una buena propuesta para programas multimedia y alcanza muy buenos resultados para programas numéricos.

En lo que al tamaño de los registros vectoriales se refiere, consideramos que, aunque en general se obtienen los mejores resultados de rendimiento con registros vectoriales de 128 elementos, la decisión final depende del número de transistores disponibles. Una implementación de propósito general debería utilizar registros de 128 elementos para poder así alcanzar un mejor rendimiento. Sin embargo, una aproximación con un coste menor puede sacrificar parte del rendimiento obtenido para disminuir así el coste de implementación. En ese caso se incluirían registros vectoriales de 16 elementos.

Los trabajos futuros se enmarcan en los temas siguientes:

- Estudio del coste de implementación de la arquitectura ILP+DLP siguiendo un modelo preciso de estimación que permita calcular los requerimientos de área de la arquitectura. Determinar la configuración de coste mínimo, la configuración de consumo mínimo y la que produzca una mejor relación coste/rendimiento.
- Estudiar modelos de memoria adicionales que mejoren el rendimiento de los accesos a memoria con stride diferente de 1, así como los accesos a memoria tipo gather/scatter.
- Entender cómo se comporta una arquitectura tipo VLIW dentro de las regiones D, y los méritos relativos de la ejecución VLIW frente a la ejecución en la arquitectura ILP+DLP.
- Estudiar el espectro de configuraciones modestas de la arquitectura ILP+DLP, en el que se puede obtener un rendimiento aceptable con un coste reducido. Estas configuraciones modestas se consiguen con un procesador ILP+DLP que posea una unidad vectorial con registros cortos.
- Estudiar el efecto de la ejecución en orden o fuera de orden en los procesadores ILP+DLP. Aunque un procesador ILP+DLP que realiza ejecución en orden de las

instrucciones es mucho más limitado, su coste también es menor, así que la relación coste/rendimiento puede ser la apropiada para la ejecución de aplicaciones con un rendimiento razonable, a la vez que se mantiene un coste reducido.

- Estudiar el rendimiento de un procesador ILP+DLP que incluya multithreading simultáneo y un sistema de memoria basado en la cache vectorial introducida en esta tesis. El estudio estaría restringido a un conjunto de configuraciones que puedan ser realmente construídas, lo cual implica un núcleo superescalar modesto, una unidad vectorial con vectores cortos, y una ejecución SMT también modesta, de forma que todo funcione a una frecuencia de reloj elevada.