LPA: A First Approach to the Loop Processor Architecture

Alejandro García[1], Oliverio J. Santana[2], Enrique Fernández[2], Pedro Medina[2], and Mateo Valero[13]

[1] Universitat Politècnica de Catalunya, Spain
{juanaleg,mateo}@ac.upc.edu
[2] Universidad de Las Palmas de Gran Canaria, Spain
{ojsantana,efernandez,pmedina}@dis.ulpgc.es
[3] Barcelona Supercomputing Center, Spain

Abstract. Current processors frequently run applications containing loop structures. However, traditional processor designs do not take into account the semantic information of the executed loops, failing to exploit an important opportunity. In this paper, we take our first step toward a loop-conscious processor architecture that has great potential to achieve high performance and relatively low energy consumption.

In particular, we propose to store simple dynamic loops in a buffer, namely the loop window. Loop instructions are kept in the loop window along with all the information needed to build the rename mapping. Therefore, the loop window can directly feed the execution back-end queues with instructions, avoiding the need for using the prediction, fetch, decode, and rename stages of the normal processor pipeline. Our results show that the loop window is a worthwhile complexity-effective alternative for processor design that reduces front-end activity by 14% for SPECint benchmarks and by 45% for SPECfp benchmarks.

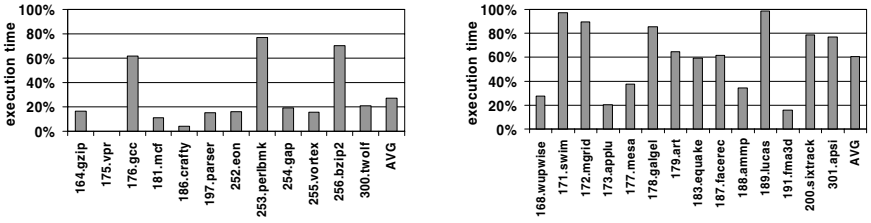
1 Introduction

Recent years have witnessed an enormous growth of the distance between the memory and the ALUs, that is, the distance between where the instructions are stored and where computation actually happens. In order to overcome this gap, current superscalar processors try to exploit as much instruction-level parallelism (ILP) as possible by increasing the number of instructions executed per cycle.

Increasing the amount of ILP available to standard superscalar processor designs involves increasing both the number of pipeline stages and the complexity of the logic required to complete instruction execution. The search for mechanisms that reduce design complexity without loosing the ability of exploiting ILP is always an interesting research field for computer architects.

In this paper, we focus on high-level loop structures. It is well known that most applications execute just 10% of their static instructions during 90% of their run time [1]. This fact is mainly due to the presence of loop structures. However, although loops are frequent entities in program execution, standard superscalar

P. Stenström et al. (Eds.): HiPEAC 2008, LNCS 4917, pp. 273–287, 2008.



(a) SPECint (b) SPECfp

Fig. 1. Percentage of time executing simple dynamic loops

processors do not have any information about whether or not the individual instructions executed belong to a loop. Indeed, when an instruction reaches the execution engine of the processor after being fetched, decoded, and renamed, it retains little or none algorithmic semantic information. Each instruction only remembers its program order, kept in a structure like the reorder buffer (ROB), as well as the basic block it belongs to support speculation.

Our objective is to introduce the semantic information of high-level loop structures into the processor. A loop-conscious architecture would be able to exploit ILP in a more complexity-effective way, also enabling the possibility of rescheduling instructions and optimizing code dynamically. However, this is not an easy design task and must be developed step by step. Our first approach to design the Loop Processor Architecture (LPA) is to capture and store already renamed instructions in a buffer that we call the loop window.

In order to simplify the design of our proposal, we take into account just simple dynamic loops that execute a single control path, that is, the loop body does not contain any branch instruction whose direction changes during loop execution. We have found that simple dynamic loops are frequent structures in our benchmark programs. Figure 1 shows the percentage of simple dynamic loops in the SPECint2000 and SPECfp2000 programs. On average, they are responsible for 28% and 60% of the execution time respectively.

The execution of a simple dynamic loop implies the repetitive execution of the same group of instructions (loop instructions) during each loop iteration. In a conventional processor design, the same loop branch is predicted as taken once per iteration. Any existing branch inside the loop body will be predicted to have the same behavior in all iterations. Furthermore, loop instructions are fetched, decoded, and renamed once and again up to all loop iterations complete. Such a repetitive process involves a great waste of energy, since the structures responsible for these tasks cause a great part of the overall processor energy consumption. For instance, the first level instruction cache is responsible for 10%–20% [2], the branch predictor is responsible for 10% or more [3], and the rename logic is responsible for 15% [4].

The main objective of the initial LPA design presented in this paper is to avoid this energy waste. Since the instructions are stored in the loop window, there is no need to use the branch predictor, the instruction cache, and the decoding logic. Furthermore, the loop window contains enough information to build the

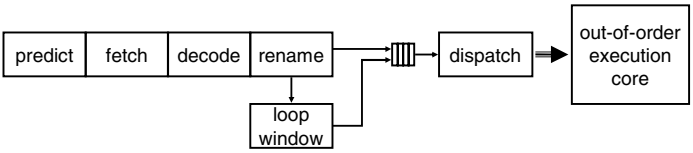


Fig. 2. LPA Architecture

rename mapping of each loop iteration, and thus there is no need to access the rename mapping table and the dependence detection and resolution circuitry.

According to our results, the loop window is able to greatly reduce the processor energy consumption. On average, the activity of the processor front-end is reduced by 14% for SPECint benchmarks and by 45% for SPECfp benchmarks. In addition, the loop window is able to fetch instructions at a faster rate than the normal front-end pipeline because it is not limited by taken branches or instruction alignment in memory. However, our results show that the performance gain achievable is limited due to the size of the main back-end structures in current processor designs. Consequently, we evaluate the potential of our loop window approach in a large instruction window processor [5] with virtually unbounded structures, showing that up to 40% performance speedup is achievable.

2 The LPA Architecture

The objective of our first approach to LPA is to replace the functionality of the prediction, fetch, decode, and rename stages during the execution of simple loop structures, as shown in Figure 2. To do this, the renamed instructions that belong to a simple loop structure are stored in a buffer that we call *loop window*. Once all the loop information required is stored in the loop window, it is able to feed the dispatch logic with already decoded and renamed instructions, making unnecessary all previous pipeline stages.

The loop window has very simple control logic, so the implementation of this scheme has little impact on the processor hardware cost. When a backward branch is predicted taken, LPA starts loop detection. All the decoded and renamed instructions after this point are then stored in the loop window during the second iteration of the loop. If the same backward branch is found and it is taken again, then LPA has effectively stored the loop. The detection of data dependences is done during the third iteration of the loop. When the backward branch is taken by the third time, LPA contains all the information it needs about the loop structure.

From this point onwards, LPA is able to fetch the instructions belonging to the loop from the loop window, and thus the branch predictor and the instruction cache are not used. Since these instructions are already decoded, there is no need to use the decoding logic. Moreover, the loop window stores enough information to build the register rename map, and thus there is no need to access the rename mapping table and the dependence detection and resolution circuitry. Therefore,

whenever LPA captures a loop, the instructions belonging to each iteration of the loop are fetched from the loop window already decoded and renamed, avoiding the need for using the prediction, fetch, decoding, and renaming logic during almost all the loop execution.

Our current LPA implementation only supports simple loop structures having a single control path inside. The appearance of an alternative path will be detected when a branch inside the loop body causes a misprediction. Therefore, the loop window is flushed at branch mispredictions, regardless the loop is still being stored or it is already being fetched from the loop window. After flushing the loop window contents, execution starts again using the normal prediction, fetch, decoding, and renaming logic.

2.1 The Renaming Mechanism

The objective of the renaming logic is to remove data dependences between instructions by providing multiple storage locations (physical registers) for the same logical register. The target register of each renamed instruction receives a physical register that is used both to keep track of data dependences and to store the value produced by the instruction. The association between logical and physical registers is kept in a structure called rename table.

Our loop window proposal is orthogonal to any prediction, fetch, and decode scheme, but it requires to decouple register renaming from physical register allocation. Instead of assigning a physical register to every renamed instruction, our architecture assigns a tag as done by virtual register [6]. This association is kept in a table that we call LVM (Logical-to-Virtual Map table). In this way, dependence tracking is effectively decoupled from value storage. Virtual tags are enough to keep track of data dependences, and thus physical register assignment is delayed until instructions are issued for execution, optimizing the usage of the available physical registers.

The LVM is a direct mapped table indexed by the logical register number, that is, it has as many entries as logical registers exist in the processor ISA. Each entry contains the virtual tag associated to the corresponding logical register. When an instruction is renamed, the LVM is looked up to obtain the virtual tags associated to the logical source registers (a maximum of two read accesses per instruction). In addition, the target register receives a virtual tag from a list of free tags. The LVM is updated with the new association between the target register and the virtual tag to allow subsequent instructions getting the correct mapping (a maximum of one update access per instruction).

Our virtual tags are actually divided into two subtags: the root virtual tag (rVT) and the iteration-dependent virtual tag (iVT). When a virtual tag is assigned to a logical register, the rVT field in the corresponding entry of the LVM receives the appropriate value from the list of free virtual tags, while the iVT field is initialized to zero. The instructions that do not belong to a captured loop will keep iVT to zero during all their execution, using just rVT for tracking dependences.

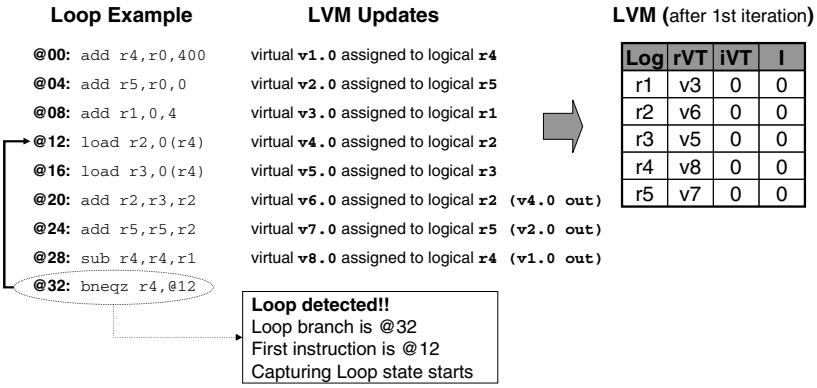


Fig. 3. Loop detection after the execution of its first iteration

The transparency of this process is an important advantage of LPA: there is no need for functional changes in the processor design beyond introducing the loop window and the two-component virtual tag renaming scheme. The out-of-order superscalar execution core will behave in the same way regardless it receives instructions from the normal pipeline or from the loop window.

2.2 Loop Detection and Storage

When a backward branch is predicted taken, LPA enters in the Capturing Loop state. Figure 3 shows an example of a loop structure that is detected by LPA at the end of its first iteration, that is, when the branch instruction finalizing the loop body is predicted taken. The backward branch is considered the loop branch and its target address is considered the first instruction of the loop body. Therefore, the loop body starts at instruction @12 and finalizes at the loop branch @32.

During the Capturing Loop state, the instructions belonging to the loop body are stored in the loop window. Data dependences between these instructions are resolved using the renaming mechanism. Figure 3 shows a snapshot of the LVM contents after the first loop iteration. We assume that there are just five logical registers in order to simplify the graph. Instructions are renamed in program order. Each instruction receives a virtual tag that is stored in the corresponding rVT field, while the iVT field is initialized to zero.

In addition, each LVM entry contains a bit (I) that indicates whether a logical register is inside a loop body and is iteration dependent. The I bit is always initialized to zero. The value of the I bit is only set to one for those logical registers that receive a new virtual register during the Capturing Loop state. An I bit set to one indicates that the associated logical register is iteration-dependent, that is, it is defined inside the loop body and thus its value is produced by an instruction from the current or the previous iteration.

Loop Example

```
@12: load r2,0(r4)
@16: load r3,0(r4)
@20: add r2,r3,r2
@24: add r5,r5,r2
@28: sub r4,r4,r1
@32: bneqz r4,@12
```

LVM Updates

virtual **v9.0** assigned to logical **r2** (v6.0 out)
virtual **v10.0** assigned to logical **r3** (v5.0 out)
virtual **v11.0** assigned to logical **r2** (v9.0 out)
virtual **v12.0** assigned to logical **r5** (v7.0 out)
virtual **v13.0** assigned to logical **r4** (v8.0 out)

LVM (after 2nd iteration)

Log	rVT	iVT	I
r1	v3	0	0
r2	v11	0	1
r3	v10	0	1
r4	v13	0	1
r5	v12	0	1

Loop Window

Instruction	@12				@16				@20				@24				@28				@32			
Operand #1	r4	v8	0	0	r4	v8	0	0	r3	v10	0	1	r5	v7	0	0	r4	v8	0	0	r4	v13	0	1
Operand #2	x	x	x	x	x	x	x	x	r2	v9	0	1	r2	v11	0	1	r1	v3	0	0	x	x	x	x
Target	r2	v9	0	1	r3	v10	0	1	r2	v11	0	1	r5	v12	0	1	r4	v13	0	1	x	x	x	x
	Log	rVP	iVP	I	Log	rVP	iVP	I	Log	rVP	iVP	I	Log	rVP	iVP	I	Log	rVP	iVP	I	Log	rVP	iVP	I

Fig. 4. Loop storage during its second iteration

Loop Example

```
@12: load r2,0(r4)
@16: load r3,0(r4)
@20: add r2,r3,r2
@24: add r5,r5,r2
@28: sub r4,r4,r1
@32: bneqz r4,@12
```

LVM Updates

virtual **v9.1** assigned to logical **r2** (v11.0 out)
virtual **v10.1** assigned to logical **r3** (v10.0 out)
virtual **v11.1** assigned to logical **r2** (v9.1 out)
virtual **v12.1** assigned to logical **r5** (v12.0 out)
virtual **v13.1** assigned to logical **r4** (v13.0 out)

LVM (after 3rd iteration)

Log	rVT	iVT	I
r1	v3	0	0
r2	v11	1	1
r3	v10	1	1
r4	v13	1	1
r5	v12	1	1

Loop Window

Instruction	@12				@16				@20				@24				@28				@32			
Operand #1	r4	v13	0	1	r4	v13	0	1	r3	v10	1	1	r5	v12	0	1	r4	v13	0	1	r4	v13	1	1
Operand #2	x	x	x	x	x	x	x	x	r2	v9	1	1	r2	v11	1	1	r1	v3	0	0	x	x	x	x
Target	r2	v9	1	1	r3	v10	1	1	r2	v11	1	1	r5	v12	1	1	r4	v13	1	1	x	x	x	x
	Log	rVP	iVP	I	Log	rVP	iVP	I	Log	rVP	iVP	I	Log	rVP	iVP	I	Log	rVP	iVP	I	Log	rVP	iVP	I

Fig. 5. Removal of dependences during the third loop iteration

Figure 4 shows a snapshot of the LVM at the end of the second iteration of our loop example, as well as the state of the loop window. All the instructions belonging to the loop body are stored in order in this buffer. Since the instructions are already decoded, the loop window should contain the instruction PC and the operation code. It should also contain the source registers, the target register, and any immediate value provided by the original instruction. In addition, each entry of the loop window should contain the renaming data for the source and target registers of the corresponding instruction, that is, the values of the rVT, iVT, and I fields of the corresponding LVM entries at the moment in which the instruction was renamed.

When the second iteration of the loop finishes and the taken backward branch is found again, then the full loop body has been stored in the loop window. However, there is not yet enough information in the loop window to allow LPA providing renamed instructions. For instance, take a look at the example in Figure 4. Instruction @32 reads the logical register $r4$. The loop window states that the value contained by this register is written by instruction @28 (virtual tag $v13.0$). This dependence is correct, since it will remain the same during all

iterations of the loop. In the next iteration of the loop, instruction @12 will also read logical register $r4$. However, the loop window does not contain the correct virtual tag value ($v9.0$ instead of $v13.0$). It happens because the loop window states that the register value depends on an instruction outside the captured loop, which was true for that iteration but not for the subsequent ones.

In order to correctly capture dependences between the instructions inside the loop body, it is necessary to execute a third iteration of the loop. Therefore, when the second iteration finishes, LPA exits the Capturing Loop state and enters the Capturing Dependences state. LPA remains in the Capturing Dependences state during the third iteration of the loop. The source operands of the instructions are renamed as usual, checking the corresponding entries of the LVM. As happened during the previous iteration, the contents of the LVM entries are also stored in the loop window. However, when the target register of an instruction requires a new virtual tag, the rVT component of the tag does not change. New virtual tags are generated increasing the value of the iVT component by one.

Figure 5 shows a snapshot of the LVM at the end of the third iteration of our loop example, as well as the state of the loop window. Now, the dependence between instruction @28 and instruction @12 is correctly stored. Instruction @28 in the second iteration generates a value that is associated to virtual tag @13.0 (Figure 4). In the current iteration, instruction @12 reads the value associated to virtual tag @13.0, while the new instance of instruction @28 generates a new value that is associated to virtual tag @13.1 and later read by instruction @32. Extending this mapping for the next iteration is straightforward, since it is only necessary to increment the iVT field by one. During the fourth iteration, instruction @12 will read the value associated to virtual tag $v13.1$ and instruction @28 will generate a value that will be associated to virtual tag $v13.2$ and later read by instruction @32.

2.3 Fetching from the Loop Window

After the third iteration finishes, the loop window contains enough information to feed the dispatch logic with instructions that are already decoded and renamed. Figure 6 shows how to generalize the information stored during previous loop iterations. Let i be the current loop iteration. The rVT values assigned to all registers remain equal during the whole loop execution. The instructions that store a value in a register during current iteration get the value i for the iVT component of the virtual tag associated to its target. Those instructions that read a value defined in the previous iteration will get the value $i-1$ for the iVT component, while those instructions that read a value defined in the current iteration will get the value i for the iVT component. Instructions that read a value defined outside the loop body will get the value 0 for the iVT component.

When an instruction is fetched from the loop window, the rVT value stored in the loop window is maintained for both the source operands and the target register. For each of these registers that has the I bit set to one, the iVT value stored in the loop window is increased by one to generate the new virtual tag that will be used in the next loop iteration. The iVT value is not increased if the

Loop Example

```
@12: load  r2, 0(r4)
@16: load  r3, 0(r4)
@20: add   r2, r3,   r2
@24: add   r5, r5,   r2
@28: sub   r4, r4,   r1
@32: bneqz r4, @12
```

Loop Mapping

```
@12: load  v9.[i],  0(v13.[i-1])
@16: load  v10.[i], 0(v13.[i-1])
@20: add   v11.[i], v10.[i],     v9.[i]
@24: add   v12.[i], v12.[i-1],   v11.[i]
@28: sub   v13.[i], v13.[i-1],   v3.0
@32: bneqz v13.[i], @12
```

Loop Window

Instruction	@12				@16				@20				@24				@28				@32			
Operand #1	r4	v13	0	1	r4	v13	0	1	r3	v10	1	1	r5	v12	0	1	r4	v13	0	1	r4	v13	1	1
Operand #2	x	x	x	x	x	x	x	x	r2	v9	1	1	r2	v11	1	1	r1	v3	0	0	x	x	x	x
Target	r2	v9	1	1	r3	v10	1	1	r2	v11	1	1	r5	v12	1	1	r4	v13	1	1	x	x	x	x
	Log	rVP	iVP	I	Log	rVP	iVP	I	Log	rVP	iVP	I	Log	rVP	iVP	I	Log	rVP	iVP	I	Log	rVP	iVP	I

Fig. 6. Generalization of the rename map obtained by LPA for our loop example

I bit value is zero, since it indicates that the value is not iteration-dependent, that is, it has been defined by an instruction outside the loop body and remains the same during all the loop execution.

For example, Figure 6 shows the state of the loop window after the third loop iteration. All the values generated during this iteration are associated to virtual tags whose iVT component value is one. During the fourth iteration, instructions @12 and @16 increment the iVT value of their source register in order to generate the correct virtual tag ($v13.1$) that allows accessing the correct value generated by instruction @28 in the previous iteration. In addition, instructions @12 and @16 store this tag in the loop window. At the end of the fourth iteration, instruction @28 generates a new value that is associated to the virtual tag obtained from increasing the iVT value stored for its target register. In the fifth iteration, instructions @12 and @16 increase again the iVT value to access the correct target of instruction @28 ($v13.2$) and so on. Meanwhile, instruction @28 always read the same value for its source register $r1$, since its I bit value is zero, and thus it does not vary during the loop execution ($v3.0$). From this point onwards, there is no change in the normal behavior of the processor shown in Figure 2. Physical registers are assigned at the dispatch stage, regardless the instructions come from the loop window or the original pipeline, and then the instructions are submitted for execution to the out-of-order superscalar back-end. In other words, LPA is orthogonal to the dispatch logic and the out-of-order superscalar execution core.

3 Experimental Methodology

The results in this paper have been obtained using trace driven simulation of a superscalar processor. Our simulator uses a static basic block dictionary to allow simulating the effect of wrong path execution. This model includes the simulation of wrong speculative predictor history updates, as well as the possible interference and prefetching effects on the instruction cache.

Table 1. Configuration of the simulated processor

fetch, rename, and commit width	6 instructions
int and fp issue width	6 instructions
load/store issue width	6 instructions
int, fp, and load/store issue queues	64 entries
reorder buffer	256 entries
int and fp point registers	160 registers
conditional branch predictor	64K-entry gshare
branch target buffer	1024-entry 4-way
RAS	32-entry
L1 instruction cache	64 KB, 2-way associative, 64 byte block
L1 data cache	64 KB, 2-way associative, 64 byte block
L2 unified cache	1 MB, 4-way associative, 128 byte block
main memory latency	100 cycles

Our simulator models a 10-stage processor pipeline. In order to provide results representative of current superscalar processor designs, we have configured the simulator as a 6-instruction wide processor. Table 1 shows the main values of our simulation setup. The processor pipeline is modeled as described in the previous section. We have evaluated a wide range of loop-window setups and chosen a 128-instruction loop window because it is able to capture most simple dynamic loops. Larger loop windows would only provide marginal benefits that do not compensate the increased implementation cost.

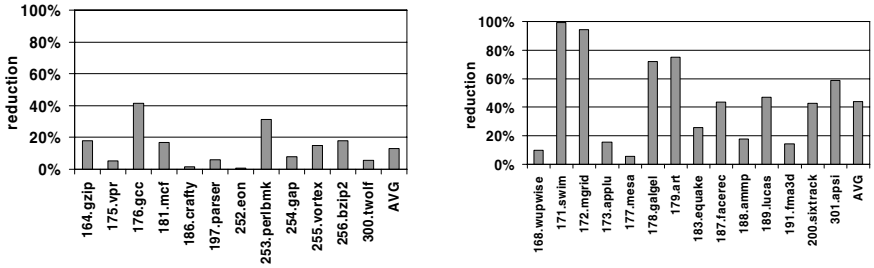
We feed our simulator with traces of 300 million instructions collected from the SPEC2000 integer and floating point benchmarks using the *reference* input set. Benchmarks were compiled on a DEC Alpha AXP 21264 [7] processor with Digital UNIX V4.0 using the standard DEC C V5.9-011, Compaq C++ V6.2-024, and Compaq Fortran V5.3-915 compilers with -O2 optimization level. To find the most representative execution segment we have analyzed the distribution of basic blocks as described in [8].

4 LPA Evaluation

In this section, we evaluate our loop window approach. We show that the loop window is able to provide great reductions in the processor front-end activity. We also show that the performance gains achievable are limited by the size of the main back-end structures. However, if an unbounded back-end is available, the loop window can provide important performance speedups.

4.1 Front-End Activity

The loop window replaces the functionality of the prediction, fetch, decode, and rename pipeline stages, allowing to reduce their activity. Figure 7 shows the activity reduction achieved for both SPECint and SPECfp 2000 benchmarks. On average, the loop window reduces the front-end activity by 14% for SPECint



(a) SPECint (b) SPECfp

Fig. 7. Reduction in the activity of the prediction, fetch, decode, and rename stages

benchmarks. This reduction ranges from less than 2% in *186.crafty* and *252.eon* up to more than 40% in *176.gcc*. The reduction is higher for the SPECfp benchmarks, since they have more simple dynamic loops. On average, SPECfp achieve 45% activity reduction, ranging from 6% in *177.mesa* to more than 95% in *171.swim* and *172.mgrid*.

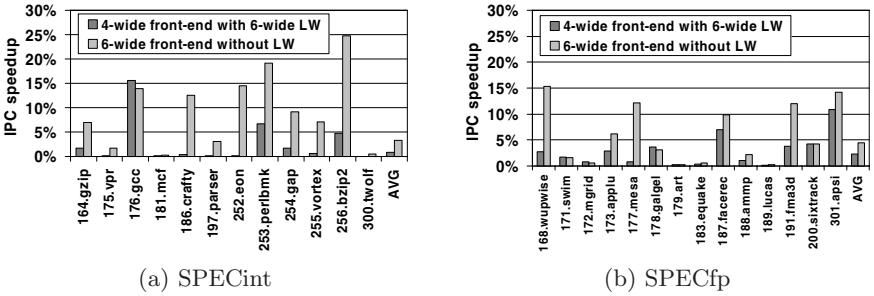
This activity reduction involves saving processor energy consumption. Reducing the total number of branch predictions involves reducing the number of accesses to the branch prediction mechanism. Reducing the number of instructions processed by the front-end involves reducing the number of accesses to the instruction cache, the decoding logic, the LVM, and the dependence check and resolution circuitry.

Although we are currently working on modeling the actual energy consumption of the processor front-end and the loop window, we are not ready yet to provide insight about this topic. Nevertheless, we are confident that the high reduction achieved in the processor front-end activity will more than compensate the additional consumption of the loop window itself, showing that the loop window is a valuable complexity-effective alternative for the design of high-performance superscalar processors.

4.2 Performance Evaluation

The loop window is able to improve processor performance because, unlike the normal front-end pipeline, it is not limited by taken branches and by the alignment of instructions in cache lines. Consequently, the loop window is able to provide instructions to the dispatch logic at a faster rate. This faster speed makes it possible to reduce the width of the processor front-end, and thus reduce its design complexity.

In this section, we evaluate the impact on performance caused by reducing the width of the processor front-end. Figure 8 shows the IPC speedup achieved by a 6-instruction wide superscalar processor (right bar) over a similar processor whose front-end is limited to just 4-instructions. As expected, increasing the front-end width from 4 to 6 instructions improves overall performance. On average, SPECint benchmarks achieve 3.3% speedup and SPECfp benchmarks



(a) SPECint (b) SPECfp

Fig. 8. Performance speedup over a 4-wide front-end processor (back-end is always 6-wide)

achieve 4.5% speedup. The improvement is larger for SPECfp benchmarks because branch instructions are better predicted, and thus it is possible to extract more ILP.

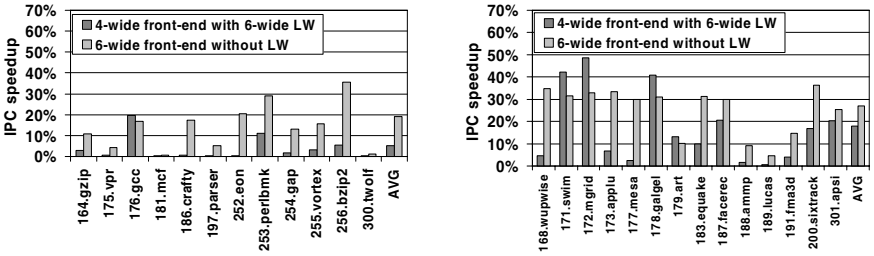
The left bar shows the speedup achieved when the front-end is still limited to 4 instructions, but a loop window able to fetch up to 6 instructions per cycle is included. This loop window allows reducing the front-end activity. In addition, a 4-instruction wide front-end is less complex than a 6-wide one. However, it becomes clear from these results that adding a loop window is not enough for the 4-wide front-end to achieve the performance of the 6-wide front-end. It only achieves comparable performance in a few benchmarks like *176.gcc*, *200.sixtrack*, and *301.apsi*. On average, SPECint benchmarks achieve 1% IPC speedup and SPECfp achieve 2.3% speedup.

The loop window is not able to reach the performance of a wider processor front-end because the most important back-end structures are completely full most of the time, that is, back-end saturation is limiting the potential benefit of our proposal. Figure 9 shows performance speedup for the same setups previously shown in Figure 8, but using an unbounded back-end, that is, the ROB, the issue queues, and the register file are scaled to infinite.

The loop window achieves higher performance speedups for SPECint and especially SPECfp benchmarks. Furthermore, the loop window using a 4-wide front-end achieves better performance than the 6-wide front-end in several benchmarks: *176.gcc* (SPECint), *172.mgrid*, *178 galgel*, and *179.art*. Those benchmarks have a high amount of simple dynamic loops, enabling the loop window to fetch instructions at a faster rate than normal front-end most of time.

5 Related Work

To exploit instruction level parallelism, it is essential to have a large window of candidate instructions available to issue. Reusing loop instructions is a well-known technique in this field, since the temporal locality present in loops provides a good opportunity for loop caching. Loop buffers were developed in the sixties for the CDC 6600/6700 series [9] to minimize the time wasted due to conditional



(a) SPECint (b) SPECfp

Fig. 9. Performance speedup over a 4-wide front-end processor using an unbounded 6-wide back-end

branches, which could severely limit performance in programs composed of short loops. The IBM/360 model 91 introduced the loop mode execution as a way of reducing the effective fetch bandwidth requirements [10,11]. When a loop is detected in an 8-word prefetch buffer, the loop mode activates. Instruction fetch from memory is stalled and all branch instructions are predicted to be taken. On average, the loop mode was active 30% of the execution time.

Nowadays, hardware-based loop caching has been mainly used in embedded systems to reduce the energy consumption of the processor fetch engine. Lee et al. [12] describe a buffering scheme for simple dynamic loops with a single execution path. It is based on detecting backward branches (loop branches) and capturing the loop instruction in a direct-mapped array (loop buffer). In this way, the instruction fetch energy consumption is reduced. In addition, a loop buffer dynamic controller (LDC) avoids penalties due to loop cache misses. Although this LDC only captures simple dynamic loops, it was recently improved [13] to detect and capture nested loops, loops with complex internal control-flow, and portions of loops that are too large to fit completely in a loop cache. This loop controller is a finite machine that provides more sophisticated utilization of the loop cache.

Unlike the techniques mentioned above, our mechanism is not only focused on reducing the energy consumption of the fetch engine. The main contribution of LPA is our novel rename mapping building algorithm, which makes it possible for our proposal to reduce the consumption of the rename logic, which is one of the hot spots in processor designs. However, there is still room for improvement. The implementation presented in this paper only captures loops with a single execution path. However, in general-purpose applications, 50% of the loops has variable-dependent trip counts and/or contains conditional branches in their bodies. Therefore, future research effort should be devoted to enhance our renaming model for capturing more complex structures in the loop window.

Although our mechanism is based on capturing simple loops that are mostly predictable by traditional branch predictors, improving loop branch prediction would be beneficial for some benchmarks with loop branches that are not so predictable. In addition, advanced loop prediction would be very useful to enable LPA to capture more complex loop patterns. Many mechanisms have been

developed to predict the behavior of loops. Sherwood et al. [14] use a Loop Termination Buffer (LTB) to predict patterns of loop branches (backward branches). The LTB stores the number of times that a loop branch is taken and a loop iteration counter is used to store the current iteration of the loop. Alba and Kaeli [15] use a mechanism to predict several loop characteristics: internal control flow, number of loop visits, number of iterations per loop visit, dynamic loop body size, and patterns leading up to the loop visit. Any of these techniques can be used in conjunction with our loop window to improve its efficiency.

Regarding our rename map building algorithm, there are other architectural proposals that try to remove instruction dependences without using a traditional renaming logic. Dynamic Vectorization [16,17] detects and captures loop structures like LPA does. This architecture saves fetch and decode power by fetching already decoded instructions from the loop storage. Rename information is also reused but it relies on a trace-processor implementation to achieve this goal. Those registers that are local to loops are renamed to special-purpose register queues, avoiding the need for renaming them in the subsequent iterations. However, only local registers take advantage of this mechanism. The live-on-entry and live-on-exit registers are still renamed once per iteration, since they are hold on a global register file using global mapping tables.

The Execution Cache Microarchitecture [18] is a more recent proposal that has some resemblance with Dynamic Vectorization. This architecture stores and reuses dynamic traces, but they are later executed using a traditional superscalar processor core instead of a trace-processor. Rename information is also reused by this architecture. However, like Dynamic Vectorization, it does not use a traditional register file but a rotating register scheme that assigns each renamed register to a register queue.

The main advantage of LPA is that it does not require such a specialized architecture. LPA can be applied to any traditional register file design. Therefore, the design and implementation cost of our proposal would be lower. In addition, all the optimization techniques developed for previous architectures like Dynamic Vectorization can be applied orthogonally. This will be an important research line to improve the performance-power features of LPA in a near future.

6 Conclusions

The Loop Processor Architecture (LPA) is focused on capturing the semantic information of high-level loop structures and using it for optimizing program execution. Our initial LPA design uses a buffer, namely the loop window, to dynamically detect and store the instructions belonging to simple dynamic loops with a single execution path. The loop window stores enough information to build the rename mapping, and thus it can feed directly the instruction queues of the processor, avoiding the need for using the prediction, fetch, decode, and rename stages of the normal processor front-end.

The LPA implementation presented in this paper reduces the front-end activity, on average, by 14% for the SPECint benchmarks and by 45% for the

SPECfp benchmarks. However, the performance gain achievable by the loop window is seriously limited by the size of the back-end structures in current processor designs. If an unbounded back-end with unlimited structures is available, the loop window achieves better performance than a traditional front-end design, even if this front-end is wider. The speedup achieved by some benchmarks like *171.swim*, *172.mgrid*, and *178.galgel* is over 40%, which suggests that the loop window could be a worthwhile contribution to the design of future large instruction window processors [5].

These results show that even the simple LPA approach presented in this paper can improve performance and, especially, reduce energy consumption. Furthermore, this is our first step towards a comprehensive LPA design that extracts all the possibilities of introducing the semantic information of loop structures into the processor. We plan to analyze loop prediction mechanisms and implement them in conjunction with the loop window. In addition, if the loop detection is guided by the branch predictor, the loop window can be managed in a more efficient way, reducing the number of insertions required and optimizing energy consumption.

The coverage of our proposal is another interesting topic for research. We only capture now simple dynamic loops with a single execution path, but we will extend our renaming scheme to enable capturing more complex loops that include hammock structures, that is, several execution paths. Increasing coverage will also benefit the possibility of applying dynamic optimization techniques to the instructions stored in the loop window, and especially those optimizations focused on loops, improving the processor performance. In general, we consider LPA is a worthwhile contribution for the computer architecture community, since our proposal has a great potential to improve processor performance and reduce energy consumption.

Acknowledgements

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN2007-60625, the HiPEAC European Network of Excellence, and the Barcelona Supercomputing Center. We would like to thank the anonymous referees for their useful reviews that made it possible to improve our paper. We would also like to thank Adrián Cristal, Daniel Ortega, Francisco Cazorla, and Marco Antonio Ramírez for their comments and support.

References

1. de Alba, M.R., Kaeli, D.R.: Runtime predictability of loops. In: Proceedings of the 4th Workshop on Workload Characterization (2001)
2. Badulescu, A., Veidenbaum, A.: Energy efficient instruction cache for wide-issue processors. In: Proceedings of the International Workshop on Innovative Architecture (2001)

3. Parikh, D., Skadron, K., Zhang, Y., Barcella, M., Stan, M.: Power issues related to branch prediction. In: Proceedings of the 8th International Symposium on High-Performance Computer Architecture (2002)
4. Folegnani, D., González, A.: Energy-effective issue logic. In: Proceedings of the 28th International Symposium on Computer Architecture (2001)
5. Cristal, A., Santana, O., Cazorla, F., Galluzzi, M., Ramírez, T., Pericàs, M., Valero, M.: Kilo-instruction processors: Overcoming the memory wall. IEEE Micro 25(3) (2005)
6. Monreal, T., González, J., González, A., Valero, M., Viñals, V.: Late allocation and early release of physical registers. IEEE Transactions on Computers 53(10) (2004)
7. Gwennap, L.: Digital 21264 sets new standard. Microprocessor Report 10(14) (1996)
8. Sherwood, T., Perelman, E., Calder, B.: Basic block distribution analysis to find periodic behavior and simulation points in applications. In: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (2001)
9. Thornton, J.E.: Parallel operation in the Control Data 6600. In: Proceedings of the AFIPS Fall Joint Computer Conference (1964)
10. Tomasulo, R.M.: An efficient algorithm for exploiting multiple arithmetic units. IBM Journal of Research and Development 11(1) (1967)
11. Anderson, D.W., Sparacio, F.J., Tomasulo, R.M.: The IBM System/360 model 91: Machine philosophy and instruction-handling. IBM Journal of Research and Development 11(1) (1967)
12. Lee, L.H., Moyer, W., Arends, J.: Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In: International Symposium on Low Power Electronics and Design (1999)
13. Rivers, J.A., Asaad, S., Wellman, J.D., Moreno, J.H.: Reducing instruction fetch energy with backward branch control information and buffering. In: International Symposium on Low Power Electronics and Design (2003)
14. Sherwood, T., Calder, B.: Loop termination prediction. In: Proceedings of the 3rd International Symposium on High Performance Computing (2000)
15. de Alba, M.R., Kaeli, D.R.: Path-based hardware loop prediction. In: Proceedings of the International Conference on Control, Virtual Instrumentation and Digital Systems (2002)
16. Vajapeyam, S., Mitra, T.: Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In: Proceedings of the 24th International Symposium on Computer Architecture (1997)
17. Vajapeyam, S., Joseph, P.J., Mitra, T.: Dynamic vectorization: A mechanism for exploiting far-flung ILP in ordinary programs. In: Proceedings of the 24th International Symposium on Computer Architecture (1999)
18. Talpes, E., Marculescu, D.: Execution cache-based microarchitectures for power-efficient superscalar processors. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 13(1) (2005)