DCache Warn: an I-Fetch Policy to Increase SMT Efficiency

Francisco J. Cazorla, Alex Ramirez, Mateo Valero
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Jordi Girona 1-3, D6
08034 Barcelona, Spain
{fcazorla, aramirez, mateo}@ac.upc.es

Enrique Fernández
Universidad de Las Palmas de Gran Canaria
Edificio de Informática y Matemáticas
Campus Universitario de Tafira
35017 Las Palmas de G.C., Spain
efernandez@dis.ulpgc.es

Abstract

Simultaneous Multithreading (SMT) processors increase performance by executing instructions from multiple threads simultaneously. These threads share the processor's resources, but also compete for them. In this environment, a thread missing in the L2 cache may allocate a large number of resources for a long time, causing other threads to run much slower than they could.

To prevent this problem we should know in advance if a thread is going to miss in the L2 cache. L1 misses are a clear indicator of a possible L2 miss. However, to stall a thread on every L1 miss is too severe, because not all L1 misses lead to an L2 miss, and this would cause an unnecessary stall and resource under-use. Also, to wait until an L2 miss is declared and squash the thread to free up the allocated resources is too expensive in terms of complexity and re-executed instructions.

In this paper we propose a novel fetch policy, which we call DWarn. DWarn uses L1 misses as indicators of L2 misses, giving higher priority to threads with no outstanding L1 misses. DWarn acts on L1 misses, before L2 misses happen in a controlled manner to reduce resource under-use and to avoid harming a thread when L1 misses do not lead to L2 misses. Our results show that DWarn outperforms previously proposed policies, in both throughput and fairness, while requiring fewer resources and avoiding instruction re-execution.

1 Introduction

Multithreaded and Simultaneous multithreaded (SMT) architectures have demonstrated to be very promising for achieving high performance by better using processor resources [5][12][13][14], with a moderate area overhead over a superscalar processor [2][6][9]. In an SMT processor several threads run together with the objective of increasing available instruction level parallelism (ILP). Co-scheduled threads use exclusively some machine resources, like the reorder buffer (ROB), and share others like the issue queues, the functional units, and the physical registers. Shared resources are dynamically allocated between threads competing for them. The instruction fetch (I-Fetch) policy determines how shared resources are filled, hence it directly affects processor performance.

An inappropriate use of the shared resources by a thread can seriously damage the performance of the remaining threads. An example of inadequate resource utilization occurs when a load misses in the L2 cache. In this case, instructions later in the thread hold shared resources for a long time while making little progress. Each instruction occupies a ROB entry and a physical register from the rename stage to the commit stage. It uses an entry in the issue queues while any of its operands is not ready, and also requires a functional unit (FU). However, neither the ROB, nor the FUs represent a problem, because the ROB is not shared and the FUs are pipelined. The actual problems are the issue queues and the physical registers, because they are used for a variable, long period. Thus, the I-fetch policy must prevent an inappropriate use of these shared resources to avoid performance degradation.

If the processor would know in advance which loads are going to miss in the L2 cache this problem would be solved. A clear in-advance indicator of L2 misses are L1 misses, because a load only misses in L2 if it previously missed in L1. Data Gating policy (DG) [3] is based on this fact, and stalls a thread every time it has an L1 miss. However, not all L1 misses cause an L2 miss. Our results show that for memory bounded threads less than 50% of L1 misses cause an L2 miss. Thus, to stall a thread every time it experiences an L1 miss is too severe and causes resource under-use. On the other hand, to react when the L2 miss is declared only alleviates the problem, because the problem still persists until then, what may be too late. FLUSH [11] works in this way

Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

and when the L2 miss is detected the thread it belongs to is flushed from the pipeline and fetch-stalled until the offending load is resolved. However, our results show that for memory bounded threads, the flushed instructions represent 35% of all fetched instructions. Obviously, this increases power consumption since many instructions need to be fetched several times.

The policy we propose in this paper, DWarn, also addresses this problem. The implementation of DWarn implies neither extensive hardware complexity nor additional power consumption. DWarn does not squash instructions in the pipeline. Furthermore, it adapts to pressure on resources reducing resource under-use. DWarn uses L1 data cache misses as indicators of a possible L2 miss. Threads experiencing an L1 data cache miss are given lower fetch priority than threads with no data cache misses. The key idea is to prevent the damage before it occurs, instead of waiting until an L2 miss is produced, when probably some damage has already been done. However, given that we are preventing possible damage (an L2 miss) and we are not sure that this damage will really occur (not all L1 misses cause an L2 miss), we prefer to lower the priority of threads instead of a more drastic measure like stalling threads entirely.

The remainder of this paper is structured as follows: we present related work in Section 2. In section 3 we explain our policy. Section 4 presents the experimental environment. Sections 5 and 6 present the results. Finally, conclusions are explained in Section 7.

2 Related work

Tullsen *et al.* [12] observe that the throughput is quite sensitive to the I-Fetch policy and propose several policies, from which ICOUNT achieves the best results. ICOUNT is defined by two parameters ICOUNT.x.y. Where x indicates the number of threads that can be asked for instructions each cycle, and y indicates the maximum number of instructions that can be fetched each cycle. ICOUNT prioritizes threads with fewer instructions in the pre-issue stages, and presents good results for threads with high ILP. However, SMT has difficulties with threads that experience many loads that miss in L2. When this situation happens, then ICOUNT does not realize that a thread can be blocked on an L2 miss and will not make progress for many cycles. When a load misses in L2, dependent instructions occupy the issue queues for a long time. On the one hand, if the number of dependent instructions is high, this thread will receive low priority. However, these entries cannot be used by other threads, degrading their performance. On the other hand, if the number of dependent instructions after a load missing in L2 is small, the number of instructions in the queues is also small. Hence this thread will receive high priority and will execute many instructions that will not be committed for a long time. As a result, the processor may run out of registers. Therefore, ICOUNT only has a limited control over the issue queues and ignores the occupancy of the physical registers.

2.1 Classifying current policies focused on the long latency problem

The performance of fetch policies dealing with load miss latency depends on the following two factors: the detection moment (DM) and the response action (RA). The DM indicates the moment in which the policy detects a load that fails or is predicted to fail in cache. Possible values range from the fetch of the load until the moment that the load finally fails in the L2 cache. Two characteristics associated with the DM are the *reliability* and the *speed*. The higher the speed of a method to detect a delinquent load, the lower its reliability. On the one hand, if we wait until the load misses in L2, we know for certain that it is a delinquent load: totally reliable but too late. On the other hand, we can predict which loads are going to miss by adding a load miss predictor to the front-end [7]. In this case, the speed is highest, but the reliability is low due to predictor mispredictions. The RA indicates the behavior of the policy once a load is detected or predicted to miss in cache, that is, it defines the measures that the fetch policy takes for delinquent threads. With these two parameters, we will classify all current policies related to long latency loads.

In [7], a mechanism called DC-PRED is presented. It uses a load miss predictor to detect the L2 missing loads in the fetch stage (we call this DM $fetch$) and for the RA, the corresponding thread is restricted to use a maximum amount of available resources ($limit\ resources$). When the missing load is resolved the thread is allowed to use all resources. The main problem of this policy is that the $fetch$ DM does not detect all loads missing in L2 and hence some loads that actually fail in the cache and that are not predicted to miss, can clog the shared resources.

In [11] two main mechanisms are propose, both using the $x\ cycles\ after\ issue$ DM. When this DM is used, a load is declared to miss in the L2 cache when it spends more cycles in the cache hierarchy than needed to access the L2 cache, including possible resource conflicts. The first mechanism flushes the instructions of the delinquent thread after the L2 missing load and also stalls the offending thread as long as the load is not resolved ($squash$ RA). We call this combination FLUSH. As a result of applying FLUSH, the offending thread temporarily does not compete for resources and, more importantly, the resources used by this thread are freed, giving the other threads full access to them. In [11], the STALL policy is also proposed. Unlike FLUSH it only stalls the offending thread without squashing it ($gate$ RA). It is less aggressive than FLUSH, does not require hardware

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

as complex as FLUSH, and does not re-execute instructions. However, its performance results are worse.

In [3], two mechanisms are proposed to reduce clog in the issue queues caused by loads that miss in the L1 data cache. The first mechanism, data gating (DG), stalls a thread when it has more than n outstanding misses. The second one, predictive data gating (PDG), is more aggressive than DG and adds an L1 data cache miss predictor. PDG stalls a thread when the number of loads predicted to miss plus the number of loads predicted to hit that in reality miss, is higher than n. In [3] the authors use the value $n = 0$, hence with DG a thread is stalled on each L1 miss. That is, it uses the $l1$ DM and the $gate$ RA. With PDG threads are stalled on each predicted miss ($fetch$ DM and $gate$ RA). The main problem of the DG and PDG policies arises when there are few threads, because the exposed parallelism and the pressure on resources are low. Consequently, to stall a thread every time it has an L1 data miss may cause an under-utilization of the resources.

As all previous policies, our DWarn policy is built on top of ICOUNT. Hence it achieves good results for ILP threads, even outperforming ICOUNT, and, in addition, it improves ICOUNT for memory bounded threads. Furthermore, DWarn does not squash instructions from the pipeline like FLUSH; it is not predictive, like PDG; and it adapts to pressure on resources more efficiently than DG and STALL.

3 The Dcache Warn policy (DWarn)

Our DWarn policy is designed to prevent the negative effects caused by loads that miss in the L2 cache and to reduce the resource under-use. DWarn uses the $l1$ DM and defines a new RA, namely, $reduce\ priority$ (see the last row in Table 1). There are two main reasons why we use the $l1$ DM. First, unlike the $fetch$ DM, the $l1$ DM detects all loads that miss in L2, because a load misses in the L2 cache if and only if it has previously missed in the L1 cache. Second, to wait X cycles after the issue of the load to take measures for the delinquent thread may be too late, because during that time the delinquent thread can have allocated many resources. In conclusion, the $l1$ DM is both reliable and early enough, avoiding that resources are clogged before the RA is carried out.

The novel $reduce\ priority$ RA is based on the combination of two ideas, namely, classification of threads, and prioritization of threads instead of gating threads.

- Classification: at each cycle available threads are classified into two groups: the first group, called Dmiss group, contains those threads that have one or more in-flight L1 data cache misses. The remaining threads belong to the second group, called Normal group. Once this dynamic classification is done, we know which threads are less-promising (Dmiss) to fetch from. We consider them to be less-promising because instructions after a missing load are more likely to be in the processor for a longer time than those instructions belonging to a thread with no in-flight data cache misses.

- Prioritization: once the classification is done, the fetch priority of the less-promising (Dmiss) threads is reduced. This is done by prioritizing fetch from Normal threads, and fetching instructions from the Dmiss threads only if there are not enough available instructions from the Normal threads. This situation happens, for example, when all Normal threads experience an instruction cache miss, or when there is only one Normal thread available. Threads in each group are sorted using ICOUNT.

DM / RA	FETCH	L1	X cycles after load issue	L2
GATE	PDG	DG	STALL	
SQUASH			FLUSH	
LIMIT RESOURCES	DC-PRED			
REDUCE PRIORITY		DWARN		

Table 1. Detection Moment-Response Action space

By reducing the priority of the Dmiss threads, the opportunity of keeping the fetch bandwidth fully used is given to Normal (more-promising) threads. Threads are never stalled, and hence, even if a thread is in the Dmiss group, it has some opportunity to fetch instructions. The idea behind this is that not all L1 misses cause an L2 miss. The fourth column of Table 2(a) shows the percentage of L1 data misses that cause an L2 miss. For MEM workloads less than 50% of L1 misses cause an L2 miss (except for the mcf benchmark). Hence, to stall a thread on each L1 data miss would be too strict a measure. However, when 2-thread workloads are executed, especially the 2-MIX and the 2-MEM workloads, very specific problems arise, which the $reduce\ priority$ RA cannot prevent. As stated in [1] the fetch fragmentation due to branches and cache line boundaries reduces fetch efficiency. Consequently, assuming that we fetch instructions from 2 threads each cycle and that the Normal thread cannot fully use the fetch bandwidth, when there are one Dmiss thread and one Normal thread, the Dmiss thread can still fetch instructions into the processor, even when it has the minimum priority. As a result, many instructions from the Dmiss thread are fetched into the processor. Little by little, internal resources are frozen

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

by these instructions, which will remain in the processor for a long time thereby degrading overall performance.

Ultimately, to harness an 8-instruction-wide processor we need either to fetch instructions from several threads in a single cycle, or use new fetch organizations that provide high fetch performance for 1.X instruction fetch policies [4]. In the former case, when we use a 2.8 fetch mechanism and only 2 threads are executed, the issue queue entries and physical registers can be occupied for a long time by threads that make little or no progress. The main reason is that the classification made by DWarn is not effective because even when the Dmiss threads are given low priority, the processor often fetches instructions from them[1].

Different workloads present different behavior depending on the number of threads and their cache miss rate. Knowing the behavior of a thread at runtime is not easy, but the number of running threads is known by the processor at any time. Hence, a way of addressing this variable behavior consists of using an I-Fetch policy that varies its behvaior based on the number of executing threads. We propose a hybrid mechanism that uses different RAs based on this number. If there are less than three threads running we will use two RAs. When a load misses in L1 the priority of its (Dmiss) thread is reduced. After that, if the load finally misses in L2 its thread is gated. In this way we prevent the Dmiss thread from clogging shared resources. If the number of execution threads is higher than 2 we will only reduce the priority of the Dmiss threads, because this is enough to prevent Dmiss threads from clogging the shared resources.

Additional hardware required to implement this technique consists of a data miss counter for each hardware context. These counters are incremented every time a thread experiences a data cache miss and decremented when the data cache fill occurs. If the counter of a thread is zero this thread will belong to the Normal group, otherwise it will belong to the Dmiss one.

4 Simulation methodology

We use a trace driven SMT simulator, based on SMTSIM [13]. The simulator consists of our own trace driven front-end and an improved version of SMTSIM's back-end. The simulator allows the execution of wrong path instructions by using a separate basic block dictionary that contains all the static instructions.

The baseline configuration is shown in Table 3, which represents a 9-stage-deep pipeline. Two important issues related to fetch policies are the following. First, the fetch is aware that a load has missed in the L1 cache 5 cycles after this load is fetched into the processor (if no resource conflicts happen). Second, it takes 10 cycles more from the L1 data miss to access the L2 cache (again, if no resource conflicts happen).

Processor Configuration	
Fetch /Issue /Commit Width	8
Fetch Policy	ICOUNT 2.8
Queues Entries	32 int, 32 fp, 32 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	384 int, 384 fp
ROB Size / thread	256 entries
Branch Predictor Configuration	
Branch Predictor	2048 entries gshare
Branch Target Buffer	256 entry, 4-way associative
RAS	256 entries
Memory Configuration	
Icache, Dcache	64K bytes, 2-way, 8-banks, 64-byte lines, 1 cycle access
L2 cache	512K bytes, 2-way, 8-banks, 10 cycles lat., 64-byte lines
Main Memory latency	100 cycles
TLB miss penalty	160 cycles

Table 3. Baseline configuration

Traces are collected of the most representative 300 million instruction segment following the idea presented in [10]. The workload consists on all programs from the SPEC2000 integer benchmark suite. Each program is executed using the reference input set and compiled with the $-O2 - non\_shared$ options using DEC Alpha AXP-21264 C/C++ compiler. Programs are divided in two groups based on their cache behaviour (see Table 2 (a)): those with an L2 cache miss rate higher than 1%[2] are considered memory bounded (MEM), the rest is considered ILP.

In [11] STALL and FLUSH are evaluated for 2-, and 4-thread workloads. In [3] DG and PDG are evaluated only for 8-thread workloads. Properties of workloads vary depending on the number of threads they have. In addition, their properties also depend on the cache behavior of the threads. In order to make a broad and fair comparison of the policies, and to avoid that our results are tuned for a special workload, we have created 12 workloads, as shown in Table 2 (b). These workloads range from 2 to 8 threads. In the ILP workloads all benchmarks have good cache behavior. All benchmarks in the MEM workloads have an L2 miss rate higher than 1%. Finally, the MIX workloads include ILP threads as well as MEM threads. For MEM workloads some benchmarks are used twice, because there are not enough SPECINT benchmarks with bad cache behavior. The replicated benchmarks are set in boldface in Table 2(b). We have shifted second instances of replicated benchmarks by one million instructions to avoid that both threads access the cache hierarchy at the same time.

[1] In section 6 DWarn is evaluated for an 1.X fetch mechanism

[2] The L2 and L1 miss rate are calculated with respect to the number of dynamic loads

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

	L1 miss rate	L2 miss rate	L1→L2	Thread type
mcf	32.3	29.6	91.6	MEM
twolf	5.8	2.9	49.3	
vpr	4.3	1.9	44.7	
parser	2.9	1.0	36.0	
gap	0.7	0.7	94.0	ILP
vortex	1.0	0.3	33.3	
gcc	0.4	0.3	82.2	
perlbmk	0.3	0.1	42.7	
bzip2	0.1	0.1	97.9	
crafty	0.8	0.1	6.9	
gzip	2.5	0.1	2.0	
eon	0.1	0.0	2.1	

Num. of threads	Thread type	Benchmarks
2	ILP	gzip, bzip2
	MIX	gzip, twolf
	MEM	mcf, twolf
4	ILP	gzip, bzip2, eon, gcc
	MIX	gzip, twolf , bzip2, mcf
	MEM	mcf, twolf, vpr, parser
6	ILP	gzip, bzip2, eon, gcc crafty, perlbmk
	MIX	gzip, twolf, bzip2, mcf, vpr, eon
	MEM	mcf, twolf, vpr, parser, **mcf**, **twolf**
8	ILP	gzip, bzip2, eon, gcc crafty, perlbmk, gap, vortex
	MIX	gzip, twolf, bzip2, mcf, vpr, eon, parser, gap
	MEM	mcf, twolf, vpr, parser, **mcf**, **twolf**, **vpr**, **parser**

Table 2. From left to right: (a) cache behavior of isolated benchmarks; (b) workloads

5 Performance evaluation

In this section we compare the efficiency of DWarn with FLUSH [11], STALL [11], DG [3], and PDG [3].

Several metrics have been proposed to measure the performance of SMT processors. These metrics attempt to balance throughput and fairness [8]. In [11] the authors evaluate STALL and FLUSH using Weighted Speedup; in [3] the Harmonic mean (Hmean) of the relative IPCs [8] is used to evaluate DG and PDG. We feel that throughput, the sum of IPCs of each thread in a given workload, is a metric as valid as the previous ones. It measures the efficient use of resources. The higher the throughput, the better the use of resources. However, it is not always valid, because to increase its value we only need to give more resources to those threads with higher IPC. Hence, to make a fair comparison, in addition to throughput, we want to use a metric that balances both throughput and fairness. In [8] it is shown that Hmean gives better throughput-fairness balance than the Weighted Speedup, hence we will use Hmean.

Before showing our results, we discuss several important issues about the implementation of the STALL, FLUSH, DG and PDG policies. Concerning STALL and FLUSH, in [11] a load is supposed to miss in L2 when it spends more than 15 cycles in the memory hierarchy. We have experimented with different values for this parameter and we found that 15 presents the best overall results for our baseline architecture. STALL and FLUSH have the following three additional properties: first, a data TLB miss also triggers a stall (flush); second, a 2-cycle advance indication is received when a load returns from memory; and third, this mechanism always keeps one thread running. That is, if there is only one thread running, it is not stopped even when it experiences an L2 miss.

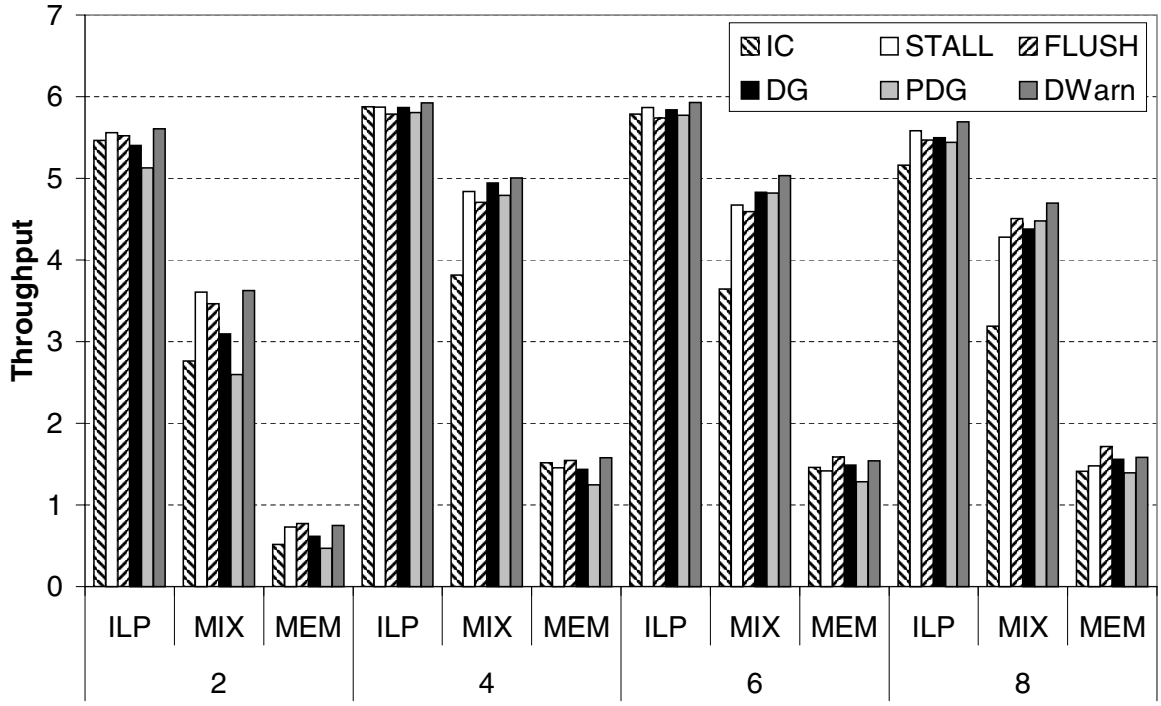
About DG, we have experimented with the number of L1 missing loads that a thread can experience before it is stalled: a low value can lead to over-stalling, a high value causes that when there are few L1 misses these are not filtered. That is, the thread they belong to is not stalled causing internal shared resources to be clogged. The value $n = 0$, the same used in [3], presents the best overall results.

5.1 Throughput results

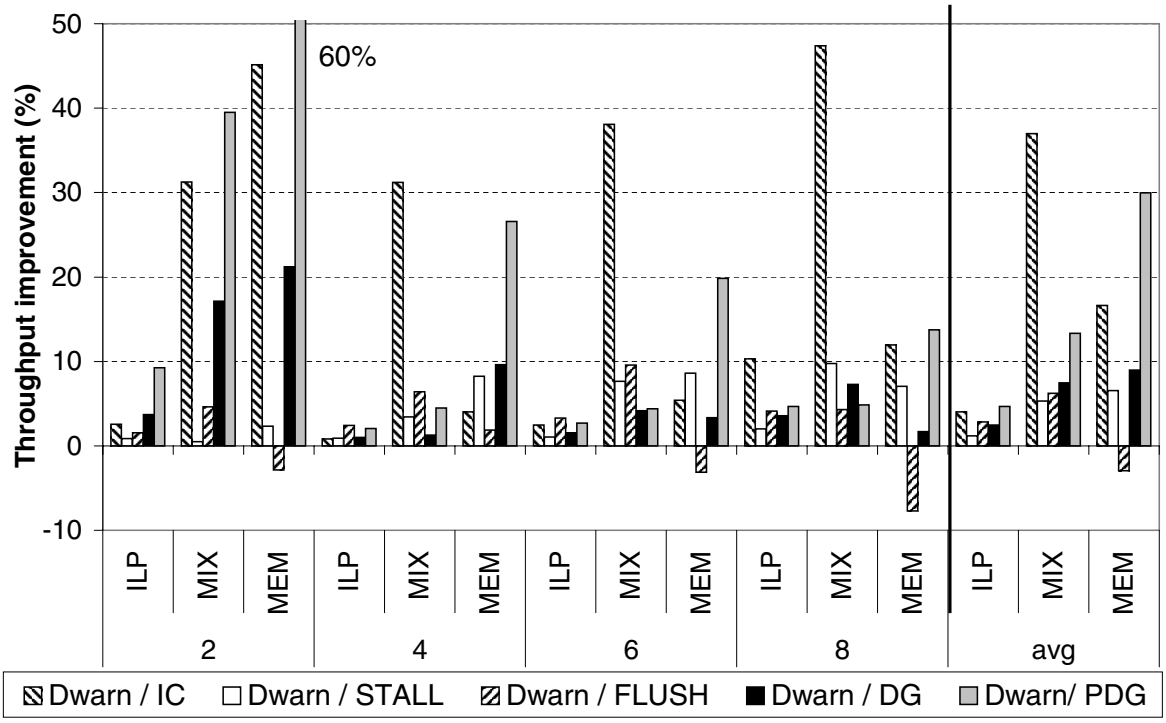
Figure 1 (a) shows the absolute IPC values for the different policies and Figure 1(b) shows the throughput improvement of DWarn over each of the others policies. It shows that DWarn outperforms all previous policies, except FLUSH for the MEM workloads. On average, DWarn outperforms ICOUNT by 18% and this improvement is higher as the number of threads increases: when there are many threads running, the pressure on shared resources is very high. Consequently, tolerating the latency of L1 misses causes a significant performance drop in the other threads. DWarn reduces the fetch priority of threads experiencing L1 misses alleviating this problem. ICOUNT does not act on an L1 miss and suffers a significant performance penalty.

Regarding DG, DWarn outperforms it for all workloads: 3% for the ILP, 8% for the MIX and 9% for the MEM workloads. This improvement gradually decreases as the number of threads increases. This is because as the number of threads increases, competition for resources also increases and then it is more difficult for a thread to allocate resources. The DG policy gates a thread on each L1 miss, so it sacrifices MEM threads in order to give more resources to ILP threads. However, if the number of threads is low, there are not enough ILP threads to use all these available resources. This means that MEM threads are unnecessarily stopped. Even worse, for MEM threads, less than 50% of L1 misses cause an L2 miss (recall the fourth column of the Table 2(a)), hence the number of unnecessary stalls is higher. With the DWarn policy no thread is stopped (for 4 or more threads), but a thread experiencing an L2 miss is executed at a lower priority. In this way, if none of the

0-7695-2132-0/04/$17.00 (C) 2004 IEEE



(a) Throughput results of the policies



(b) Throughput improvement of DWarn over the other policies

Figure 1. Throughput results

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

available Normal threads with higher priority can use the resources they are given to this thread. Thus, resource-underutilization is reduced.

Regarding PDG, it suffers the same problems as DG, but, in addition, PDG presents 2 additional problems: the first is the predictor mistakes. If the predictor erroneously indicates that a load will miss in the L1 cache and in fact the load does not, this causes an additional unnecessary stop, degrading the performance of the thread. The second problem is the load serialization. Our results (not shown here) indicate that many cache misses occur close in time. Hence, to stop on each missing load in the fetch stage causes load serialization and loss of the available memory parallelism. DWarn improves PDG by 5% for the ILP, 13% for the MIX and 30% for the MEM workloads.

Regarding STALL, DWarn improves it for all workloads. The improvements are of 2% for the ILP workloads, 6% for the MIX workloads and 7% for the MEM workloads.

DWarn also outperforms FLUSH, for the ILP and MIX workloads, by 3% and 6% respectively, and only suffers a slowdown of 3% for the MEM workloads. The main cause is that for the 6-MEM and 8-MEM workloads the pressure over resources is too high, hence it is more preferable to free resources by flushing the delinquent threads than to freeze resources by stalling these threads. However, this improvement of 3% achieved by FLUSH comes at a high cost. First, DWarn does not require such a complex hardware as FLUSH, and second, it does not re-execute instructions. As shown in Figure 2, the number of squashed instructions due to the FLUSH policy represents a significant percentage of the fetched instructions (35% for the MEM workloads).

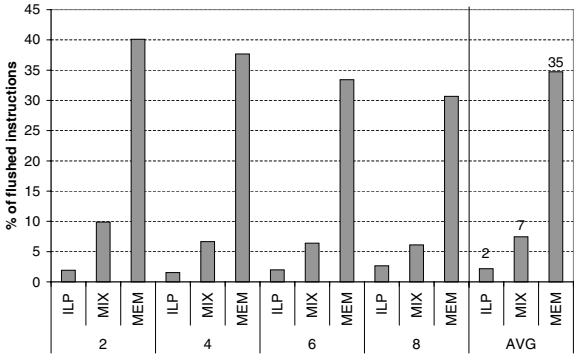


Figure 2. Flushed instructions with respect to fetched instructions (FLUSH policy)

The problem of FLUSH is that, like STALL, it reacts too late, namely, when the L2 miss has been detected, and in a drastic way by flushing all the instructions after the load. The former problem implies that as long as the L2 miss is not declared, instructions after the missing load compete for resources with instructions of other threads that could contribute to final IPC, degrading performance. The latter problem directly affects the thread being flushed, reducing its performance. DWarn acts earlier, when an L1 miss occurs, and in a controlled manner by reducing the presumed delinquent thread's priority. As a result, only when the remaining threads are not able of using machine resources, these are given to the delinquent thread and hence its performance is less affected.

	RELATIVE IPCs				Hmean
	thread 1 ILP	thread 2 ILP	thread 3 MEM	thread 4 MEM	
ICOUNT	0.36	0.41	0.50	0.79	0.47
STALL	0.42	0.65	0.38	0.63	0.49
FLUSH	0.41	0.64	0.34	0.59	0.46
DG	0.43	0.70	0.34	0.46	0.45
PDG	0.40	0.72	0.28	0.31	0.38
DWARN	0.44	0.69	0.43	0.70	0.53

Table 4. Relative IPC of each thread in the 4-MIX workload

5.2 Harmonic Mean results

The second metric we have used is the Harmonic Mean (Hmean) [8]. The Hmean metric attempts to avoid artificial improvements achieved by giving more resources to threads with high ILP.

Figure 3 depicts the Hmean improvement of DWarn policy over the other policies. Average results indicate that DWarn improves all other policies for ILP, MIX and MEM workloads, only suffering a slowdown of 2% respect to FLUSH. Let us illustrate why DWarn outperforms all previous policies by an example. Table 4 shows the relative IPC of each thread in the 4-MIX workload for all policies. The second and the third columns indicate the relative IPC of the two ILP threads in the workload. Likewise, the two following columns indicate the relative IPC of the two MEM threads. The point is that DWarn achieves an IPC for the ILP threads that is as high as the one obtained by the other policies, but it does not significantly affect the IPC of the MEM threads as the other policies do. The reason for this is that DWarn never stalls or squashes threads even when they are in the Dmiss group for four or more threads. As a result, if there are available resources and the Normal threads cannot use them, these resources are given to the Dmiss threads. Regarding ICOUNT, we observe that ICOUNT achieves the highest result for the MEM threads, but it heavily degrades the performance of the ILP threads.

Concluding, the DWarn policy presents the best balance between achieving high IPC for the ILP threads and harm-

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

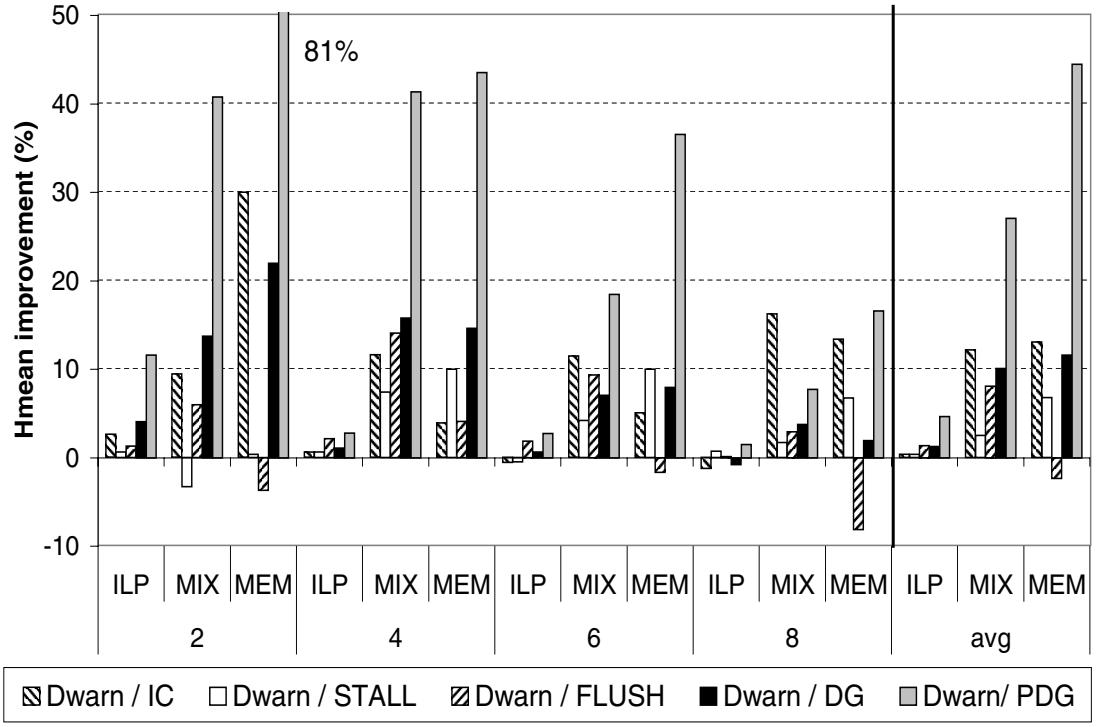


Figure 3. Hmean improvement of DWarn over the other policies

ing as little as possible the IPC of MEM threads. As a result, on average, DWarn achieves better Hmean results than the other policies for all types of workloads. It only experiences a slowdown of 2% with respect to FLUSH for the MEM workloads. However, as we saw in Figure 2, FLUSH achieves this is at the cost of increasing the number of fetched instruction by 35%.

We have seen that depending on the number of running threads, workloads present very different properties. When there are few threads, reducing priority is not enough to avoid Dmiss threads from using shared resources for a long time. This is because thread level parallelism is low and Normal threads do not use all the fetch bandwidth. When there are 6 and 8 threads, the pressure on resources is very high and hence the competition for them is high too. A general purpose I-fetch policy must be aware of the number of executing threads to better adapt to the properties of each workload. Overall results show that our hybrid mechanism, which combines the $gate$ and the $reduce\ priority$ RAs when there are two threads running, and uses the $reduce\ priority$ RA in the remaining cases, outperforms all previous proposed policies.

6 DWarn on different architectures

The time needed to determine whether a load has caused an L1 data cache miss or an L2 miss are two key factors in the previous policies. Another important factor is the number of threads that can fetch instructions into the processor each cycle. In this section, we experiment with two variants of the original architecture in order to analyze the effect of these factors.

The first architecture represents a less aggressive processor than presented in Table 3. This is a 4-wide, 4-context processor with an 1.4 fetch mechanism. There are 256 physical registers and 3 integer, 2 floating point, and 2 load/store functional units. In this architecture, we can fetch instructions only from one thread each cycle, and hence, the Dmiss threads cannot fetch if there is at least one Normal thread. On the one hand, it is beneficial because Dmiss threads can unlikely clog resources. On the other hand, MEM threads are now more damaged. Figures 4 (a) and (b) show the throughput and Hmean improvement of the DWarn policy over the other ones. Hmean results show that for the MIX workloads DWarn is outperformed by ICOUNT, by 5% on average. The main cause for this slowdown is that the MEM threads are now heavily damaged. Regarding the other policies addressing the load miss latency problem, DWarn

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

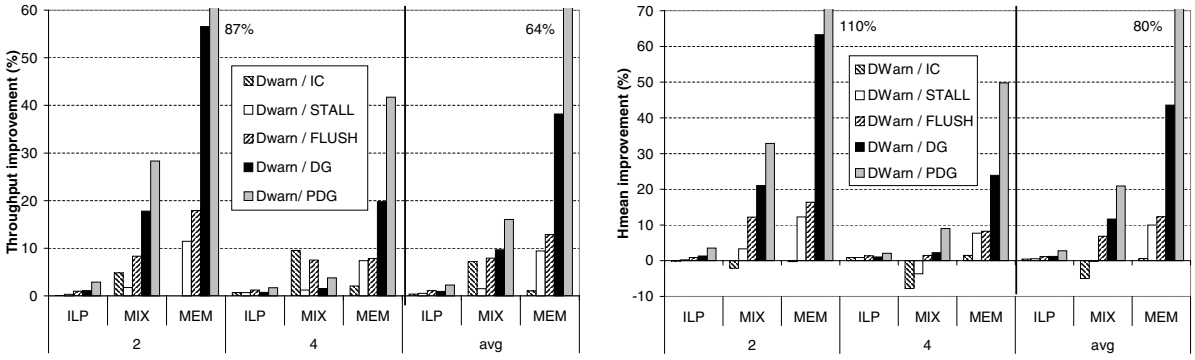


Figure 4. From left to right: (a) Throughput improvement of DWarn over the other policies; (b) Hmean improvement (smaller architecture)

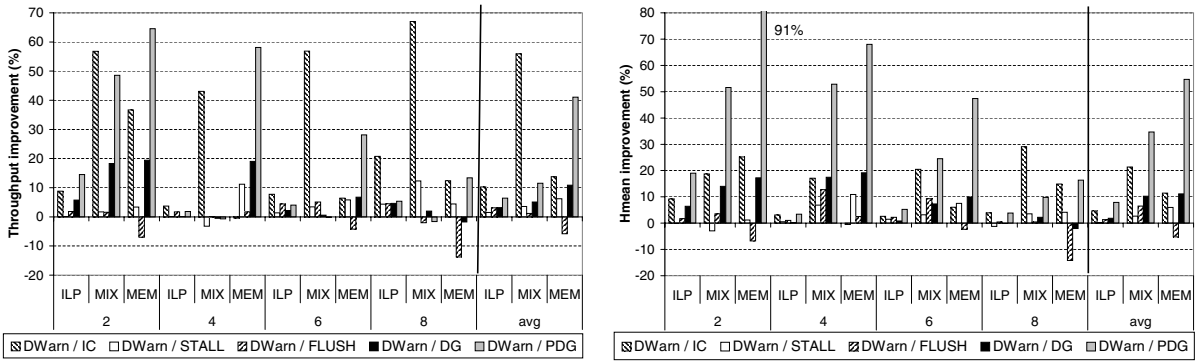


Figure 5. From left to right: (a) Throughput improvement of DWarn over the other policies; (b) Hmean improvement (deeper architecture)

clearly outperforms them in both throughput and Hmean. The throughput improvements for the MIX and the MEM workloads are: 5% over STALL, 23% over DG, 10% over FLUSH, and 40% over PDG. Hmean improvements for the MIX and the MEM workloads are: 5% over STALL, 28% over DG, 10% over FLUSH, and 50% over PDG.

The second architecture represents a deeper and more aggressive processor than presented in Table 3. This is a 16-stage-depth processor, with a 2.8 fetch mechanism, and 64-entry issue queues. The time to determine an L1 miss has been incremented by 3 cycles, the latency from the L1 cache to the L2 from 10 to 15 cycles, and the memory latency has also been incremented to 200 cycles. Figures 5 (a) and (b) show the throughput and Hmean improvement of the DWarn policy over the other ones. As we see in the average results, DWarn throughput and Hmean results indicate that it improves all other policies for all type of workloads, except for the MEM where it suffers a slowdown of 6% with respect to FLUSH. The main cause for this high average slowdown is the 8-MEM workload. In that case, there is an over-pressure on resources (our results show that the throughput for the 4-MEM workload is almost the same than for the 6-, and the 8-MEM workloads), whereby flushing is much more effective than stalling threads. However, our results (not shown here) show that this is at the cost of increasing the number of fetched instructions due to flushes. This increment is 56% on average for the MEM workloads.

7 Conclusions

The performance of an SMT processor directly depends on how the dynamic allocation of shared resources is done. The instruction fetch policy dynamically determines how this allocation is carried out. To achieve high performance, the fetch policy must avoid the monopolization of a shared resource by any thread. An example of this situation occurs

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

when a load misses in the L2 cache level. ICOUNT [12] reacts too late and suffers significant performance degradations, in particular for 2-thread workloads where this problem is most acute. In this case, FLUSH [11] clearly outperforms ICOUNT. However, the problem of FLUSH is that it does not prevent damage, but drastically cures it once it has happened. Other policies [3] try to prevent this damage by acting before the L2 miss occurs. DG stalls threads on an L1 miss and PDG on a predicted L1 miss in the fetch stage. However, when there are few threads, these policies are too strict and cause resource under-utilization and an important performance degradation.

In this paper, we propose a novel policy that deals with this problem (DWarn). DWarn is not predictive and requires minimum hardware resources. It does not flush instructions reducing overall processor complexity and wasted power. Furthermore, DWarn adapts to pressure on resources better than the other policies. If there are few running threads, it avoids resource under-utilization. When the number of threads increases, reducing fetch priority is enough to avoid Dmiss threads from holding resources for a long time.

On average, DWarn throughput results show that it improves all other policies for all types of workloads, especially for the MIX and MEM ones: 27% over ICOUNT, 6% over STALL, 2% over FLUSH, 8% over DG and 22%over PDG. DWarn only suffers a loss of 3% with respect to FLUSH for the MEM workloads. However, this comes at the cost of increasing hardware complexity for these policies and the number of fetched instructions (35% for memory bounded threads). Fairness results show that DWarn clearly presents the best throughput-fairness balance, only suffering a slowdown of 2% with respect to FLUSH for the MEM workloads. The improvement for the MIX and MEM workloads are: 13% over ICOUNT, 5% over STALL, 3% over FLUSH, 11% over DG and 36%over PDG. If we take into consideration all these results, DWarn presents as the best solution to the problem of long latency loads for throughput, fairness, complexity, and power.

Acknowledgments

This work has been supported by an Intel fellowship and the Ministry of Science and Technology of Spain under contract TIC-2001-0995-C02-01, and grant FP-2001-2653 (Francisco J. Cazorla). The authors would like to thank Oliverio J. Santana, Ayose Falcón, Fernando Latorre and Peter Knijnenburg for their comments and work in the simulation tool. The authors also would like to thank the reviewers for their valuable comments.

References

[1] J. Burns and J.-L. Gaudiot. Exploring the SMT fetch bottleneck. In *Proceedings of the 3rd Workshop on Multithreaded Execution, Architecture, and Compilation*, 1999.

[2] J. Burns and J.-L. Gaudiot. Quantifying the SMT layout overhead- does SMT pull its weight? In *Proceedings of the 6th Intl. Conference on High Performance Computer Architecture*, pages 109–120, Jan. 2000.

[3] A. El-Moursy and D. Albonesi. Front-end policies for improved issue efficiency in SMT processors. In *Proceedings of the 9th Intl. Conference on High Performance Computer Architecture*, pages 31–42, Feb. 2003.

[4] A. Falcon, A. Ramirez, and M. Valero. A low complexity, high-performance fetch unit for simultaneous multithreading processors. *Proceedings of the 10th Intl. Conference on High Performance Computer Architecture*, Feb. 2004.

[5] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proc. of the 19th Annual ISCA*, pages 136–145, May 1992.

[6] R. Kalla, B. Sinharoy, and J. Tendler. SMT implementation in POWER 5. *Hot Chips*, 15, Aug. 2003.

[7] C. Limousin, J. Sebot, A. Vartanian, and N. Drach-Temam. Improving 3D geometry transformations on a simultaneous multithreaded SIMD processor. In *Proc. of the 15th Intl. Conference on Supercomputing*, pages 236–245, May 2001.

[8] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 164–171, Nov. 2001.

[9] D. T. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), Feb. 2002.

[10] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 10th Intl. Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[11] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreaded processor. In *Proceedings of the 34th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 318–327, Dec. 2001.

[12] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23th Annual Intl. Symposium on Computer Architecture*, pages 191–202, Apr. 1996.

[13] D. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th Annual Intl. Symposium on Computer Architecture*, pages 392–403, 1995.

[14] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Proceedings of the 4th Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 49–58, June 1995.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE