Optimizing Long-Latency-Load-Aware Fetch Policies for SMT Processors

Francisco J. Cazorla, Enrique Fernandez, Alex Ramirez and Mateo Valero, *Fellow, IEEE*

***Abstract*— Simultaneous Multithreading (SMT) processors fetch instructions from several threads and, in this way, the available Instruction Level Parallelism (ILP) of each thread is exposed to the processor. In an SMT processor the fetch engine has the additional level of freedom, compared to a super-scalar processor, to select independent instructions. The fetch engine determines how shared resources are allocated, playing a key role in the final performance of the machine.**

When a thread experiences an L2 cache miss, critical resources can be monopolized for a long time, throttling the execution of remaining threads. Several approaches have been proposed to cope with this problem. The first contribution of this paper is the evaluation and comparison of the three best published policies addressing the long latency load problem. The second and main contributions of this paper are that we have proposed improved versions of these three policies. Our results show that the improved versions significantly enhance the original ones in both throughput and fairness.

***Index Terms*— SMT, multithreading, fetch policy, long latency loads, load miss predictors.**

I. Introduction

MULTITHREADED and Simultaneous Multithreaded Processors (SMT) [1][2][3][4] concurrently run several threads in order to increase the available parallelism, with a moderate area overhead over a super-scalar processor [5][6][7][8]. The sources of this parallelism come from the instruction level parallelism (ILP) in each thread alone, from the additional parallelism that provides the freedom of fetching instructions from different independent threads, and from combining them in an appropriate way. Problems arise because shared resources have to be dynamically allocated between these threads. It is the responsibility of the fetch policy to decide which instructions and from which thread come into the processor. Hence, it determines how this allocation is done, playing a key role in obtaining performance.

When a thread experiences an L2 cache miss, instructions after this load occupy resources for a long time while making little progress. Each instruction occupies a ROB entry and a physical register, not all, from the rename stage to the commit stage. It also uses an entry in the issue queues while any of its operands is not ready, and a functional unit (FU). Neither the ROB nor the FUs present a problem, because the ROB is not shared and the FUs are pipelined. The issue queues and the physical registers are the actual problem, because they are used for a variable and long period. Thus, the instruction fetch (I-fetch) policy must prevent an incorrect use of these shared resources to avoid significant performance degradation.

Several policies have been proposed to alleviate the previous problem. As far as we know, the first proposal to address it was mentioned in [9]. The authors suggest that a load miss predictor could be used to predict L2 misses, switching between threads when one of them is predicted to have an L2 miss. In [10] the authors propose two mechanisms to reduce load latency: data pre-fetching and a policy based on a load miss predictor (we explain these policies in the related work section). STALL [11] fetch-stalls a thread when it is declared to have an L2 missing load until the load is resolved. FLUSH [11] works similarly and additionally flushes the thread to which the missing load belongs. DG [12] and PDG [12] are two recently proposed policies that try to reduce the effects of L1 missing loads. Our performance results show that FLUSH outperforms both these policies, hence we will not evaluate DG and PDG in this paper.

In the first part of this paper we analyze the use of a load miss predictor in an SMT processor, and compare it with the FLUSH and STALL policies. As we show below none of them outperforms all others in all cases: each behaves better than the others depending on the metric used and on the workload. Based on this initial study, in the second part we propose improved versions of each policy. Throughput and fairness [13] results show that in general improved versions achieve significant performance improvement over the original versions for a wide range of workloads, ranging from two to eight threads.

The remainder of this paper is structured as follows: we present related work in Section 2. Section 3 presents the experimental environment and the metrics used to compare the different policies. In Section 4 we analyze the use of a load miss predictor in an SMT processor. Section 5 compares the effectiveness of those policies. In Section 6 we propose several improvements for the presented polices. In Section 7 we compare the improved policies. Finally, Section 8 is devoted to the conclusions.

II. Related Work

Current I-fetch policies address the problem of the latency of load that miss in the L2 cache in several ways. Round Robin [2] is absolutely blind to this problem. Instructions are alternately fetched from available threads, even when they

Manuscript received XXX, 2002; revised YYY; accepted ZZZ.

Francisco J. Cazorla, Alex Ramirez and Mateo Valero are with the Computer Architecture Departament, UPC, Jodi Girona 1-3 D6. 08034 Barcelona, Spain. E-mail:{fcazorla, aramirez, mateo}@ac.upc.es

Enrique Fernandez is with the Univeristy of Las Palmas de Gran Canaria. E-mail: efernandez@dis.ulpgc.es

0000–0000/00$00.00 © 2003 IEEE

have in-flight L2 misses. ICOUNT [2] only takes into account the occupancy of the issue queues and disregards that a thread can be blocked on an L2 miss, causing this thread to make no progress for many cycles. ICOUNT gives higher priority to those threads with fewer instructions in the queues and in the pre-issue stages. When a load misses in L2, dependent instructions occupy the issue queues for a long time. If the number of dependent instructions is high, this thread will have low priority. However, these entries cannot be used by the other threads, degrading their performance. Conversely, if the number of dependent instructions after a load missing in L2 is low, the number of instructions in the queues is also low, so this thread will have high priority and will execute many instructions that cannot be committed for a long time. As a result, the processor can run out of registers. Therefore, ICOUNT only has a limited control over the issue queues, because it cannot prevent threads from using the issue queues for a long time. Furthermore, ICOUNT ignores the occupancy of the physical registers.

More recent policies, implemented on top of ICOUNT, focus in this problem and add more control over issue queues, as well as control over the physical registers. In [9], a load hit/miss predictor is used in a super-scalar processor to guide the dispatch of instructions that the scheduler makes. This allows the scheduler to dispatch dependent instructions at the time they require data. The authors propose several hit/miss predictors that are adaptations of well known branch miss predictors. The authors suggest adding a load miss predictor in an SMT processor in order to detect L2 misses. This predictor would guide the instruction fetch, switching between threads when any of them is predicted to miss in L2.

In [10] the authors propose two mechanisms focused on reducing the problem associated with load latencies. They use data prefetching and conclude that it is not effective because, although the latency of missing loads is reduced, this latency is still significant. Furthermore, as the number of threads increases, the gain decreases due the pressure put on the memory bus. The second mechanism uses a load miss predictor, and when a load is predicted to miss, the corresponding thread is restricted to use a maximum amount of available resources. When the missing load is resolved, the thread is allowed to use all the resources.

In [11], the authors propose several mechanisms to detect an L2 miss (detection mechanism) and different ways of acting on a thread once it is predicted to have an L2 miss (action mechanism). The detection mechanism that presents the best results, is to predict 'miss' every time that a load spends more cycles in the cache hierarchy than needed to access the L2 cache, including possible resource conflicts (15 cycles in the simulated architecture). Two action mechanisms present good results. The first one, STALL, consists of fetch-stalling the offending thread. The second one, FLUSH, flushes the instructions after the load missing in L2, and also stalls the offending thread until the load is resolved. As a result, the offending thread temporarily does not compete for resources and, what is more important, the resources used by the offending thread are freed, giving the other threads full access to them. FLUSH requires complex hardware and increases the pressure on the front-end of the machine because it requires squashing all instruction after a missing load. Furthermore, due to the squashes, many instructions need to be re-fetched and re-executed. STALL is less aggressive than FLUSH, does not require hardware as complex as FLUSH, and does not re-execute instructions.

In this paper we present improved versions of FLUSH, STALL, and L2MP that clearly improve the original ones in both throughput and fairness.

III. Metrics and Experimental Setup

We have used three different metrics to make a fair comparison of the policies. First, the IPC throughput. Second, a metric that balances throughput and fairness (Hmean [13]). And third, a metric that takes into account the extra energy used to re-execute of instructions (extra fetch or EF).

We call the fraction IPC_{wld}/IPC_{alone} the *relative IPC*, where the IPC_{wld} is the IPC of a thread in a given workload, and the IPC_{alone} is the IPC of a thread when it runs isolated. The Hmean metric is the harmonic mean of the relative IPC of the threads in a workload . Hmean is calculated as shown in equation 1.

$$Hmean = \frac{\#threads}{\sum_{threads} \frac{IPC_{alone}}{IPC_{wld}}} \quad (1)$$

The extra fetch (EF) metric measures the extra instructions fetched due to flushing of instructions,see equation 2. Here we are not taking into account the flushed instructions due to branch mispredictions, but only those caused by loads missing in L2. EF compares the total number of fetched instructions (flushed and not flushed) with the number of instructions that are fetched and not flushed. The higher the value of the EF is, the higher the number of squashed instructions with respect to the total number of fetched instructions.

$$EF = \frac{TotalFetched * 100}{Fetched\ not\ squashed} - 100 \quad (2)$$

We use a trace driven SMT simulator, based on SMTSIM [3]. It consists of our own trace driven front-end and a modified version of SMTSIM's back-end. The baseline configuration is shown in Table I(a). The simulator allows executing wrong path instructions by using a separate basic block dictionary that contains all static instructions.

Traces have been collected of the most representative 300 million instruction segment, following an idea presented in [14]. The workload consists of all programs from the SPEC2000 integer benchmark suite. Each program was executed using the reference input set and compiled with the $-O2 -non_shared$ options, using the DEC Alpha AXP-21264 C/C++ compiler. Programs are divided in two groups based on their cache behavior, see Table I (b): those with an L2 cache miss rate higher than 1%[1] are considered memory bounded (MEM), the rest is considered ILP. From these programs we have created 12 workloads, as shown in Table II, ranging from 2 to 8 threads. In the ILP workloads all benchmarks

[1]The L2 miss rate is calculated with respect to the number of dynamic loads

TABLE I

FROM LEFT TO RIGHT. (A) BASELINE CONFIGURATION; (B) L2 BEHAVIOR OF ISOLATED BENCHMARKS

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	
L1 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

	L2 miss rate	Thread type
mcf	29.6	MEM
twolf	2.9	
vpr	1.9	
parser	1.0	
gap	0.7	ILP
vortex	0.3	
gcc	0.3	
perlbmk	0.1	
bzip2	0.1	
crafty	0.1	
gzip	0.1	
eon	0.0	

TABLE II

WORKLOADS

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, **twolf**
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**

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 were used twice, because there are not enough SPECINT benchmarks with bad cache behavior. The replicated benchmarks are highlighted in boldface in Table II. We have shifted the second instance of a replicated benchmarks by one million instructions in order to avoid that both threads accessing the cache hierarchy at the same time.

IV. LOAD MISS PREDICTORS IN AN SMT ENVIRONMENT

In this paper we use several predictors to predict L2 misses. First, we describe the policy that uses load miss predictors, what called L2MP. This policy is similar to PDG [12] but it predicts L2 misses instead of L1 misses. After that, we describe the load miss predictors that we evaluate as well as the metrics used to compare the effectiveness of the different load miss predictors.

A. L2MP policy

The L2MP scheme is shown in Figure 1. The predictor acts in the decode stage. It is a table indexed by the PC of a load: if a load is not predicted (1) to miss in L2, it executes normally. If a load is predicted (1) to miss in L2 cache, the thread it belongs to is stalled (2). This load is tagged indicating that it has stalled the thread. When this load is resolved, either in the Dcache (3), or in the L2 cache(4), the corresponding thread continues.

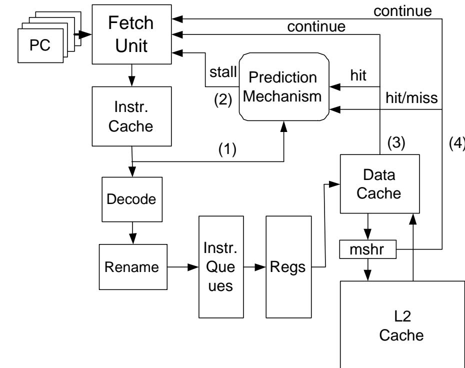


Fig. 1. L2MP mechanism

B. Load miss predictors

We have explored a wide range of load miss predictors. The one that obtains the best results is the predictor proposed in [10]. We call this predictor *pattern predictor*. In this paper, we show the results of the pattern predictor and also for the predictor proposed in [12], called 2bc.

The pattern predictor uses one table per thread. Each table is direct mapped and each of its entries contains three fields: the tag, the number of hits in the cache (Data cache and L2 cache) between the two last misses in L2, and the number of hits in the cache since the last miss. A load is predicted to miss in L2 when the last two fields are equal. The table is updated when a load hits in the cache (Data cache or L2) and when the load misses in L2.

The 2bc predictor uses only one table that is shared between threads. This table is indexed by the PC of the load and it has 2K entries of two bit saturating counters. On a miss, the

corresponding entry is cleared and on a hit, it is incremented. The most significant bit determines the prediction.

C. Metrics to compare load miss predictors

As in [9], we classify every dynamic load into one of four groups based on the result of the load (hit or miss in cache), and on the prediction made by the predictor. The contribution of this paper is that we identify and analyze the particularities of each group in an SMT processor.

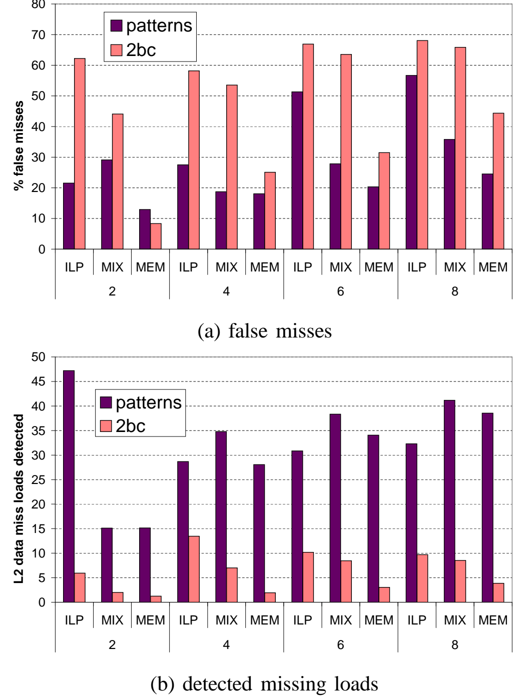
We differentiate four groups, the first two represent predictor hits, and the last two predictor misses.

- AHPH (actual hit - predicted hit): this group is formed by the loads that hit in the Data cache or in the L2 cache and that are not predicted to miss. Thus, they are executed normally.
- AMPM (actual miss - predicted miss): this group represents the loads missing in L2 that are detected by the predictor. That is, the loads correctly stalled.
- AHPM (actual hit - predicted miss): loads that hit in the L1 cache or in the L2 cache that are predicted to miss. In this case, the corresponding thread is unnecessarily fetch-stalled until this load is resolved. Hence, its performance is degraded. If the load hits in L1 data cache, it takes approximately 5 cycles in the simulated architecture to re-start the thread. If the load miss in L1 and hits in L2, it takes approximately 15 cycles in the simulated architecture.
- AMPH (actual miss - predicted hit): this group covers those loads that actually miss in L2 but that are not detected by the predictor. These loads heavily degrade the performance of the SMT processor because machine resources are clogged by the instructions after these loads.

To measure the effectiveness of the predictors we use two metrics. First, *false misses*: the percentage of incorrect mispredictions (AHPM/PM). Second, *filtered loads*: the percentage of actual misses (AM) that are detected by the predictor (AMPM/AM). The objective of the predictor is to maximize the filtered loads while minimizing the false misses. However, there is a trade-off between these two factors, because to achieve a high percentage of filtered loads the predictor must be more aggressive probably increasing the false misses.

D. Load miss predictors evaluation

Concerning to the pattern predictor, in [10] authors do not indicate the size of the predictor. We have explored different sizes, and the best result is obtained when the table has 8 entries. In this case the total predictor size is 2.944 Kbits (8 threads x 8 entries x (30 bits of tag + 8 bit counter + 8 bit counter)). Figures 2(a) and (b) compare the effectiveness of both predictors. These figures respectively show the percentage of false misses and the percentage of detected L2 missing loads. We observe that the pattern predictor presents the best results: it has less false misses and detects more L2 missing loads. Hence, we use this predictor in the L2MP policy in the remainder of this paper.



(a) false misses

(b) detected missing loads

Fig. 2. Effectiveness of the pattern predictor and the 2bc

V. Comparing the Current Policies

In this section we determine the effectiveness of the different policies addressing the problem of load latency. We compare the STALL, FLUSH and L2MP policies using the throughput and the Hmean metrics. In Figure 3, we show the throughput and the Hmean improvements of STALL, FLUSH, and L2MP over ICOUNT. L2MP achieves important throughput improvements over ICOUNT, mainly for 2-thread workloads. However, fairness results using the Hmean metric indicate that for the MEM workloads the L2MP is more unfair than ICOUNT. Only for 8-thread workloads L2MP outperforms ICOUNT in both throughput and Hmean. Our results indicate that this is because L2MP hurts MEM threads and boosts ILP threads, especially for few-thread workloads. If we compare the effectiveness of L2MP to other policies addressing the same problem, like STALL, we observe that L2MP only achieves better throughput than STALL for MEM workloads. However, L2MP heavily affects fairness. We explain why L2MP does not obtain results as good as STALL below.

The results of FLUSH and STALL are very similar. In general FLUSH slightly outperforms STALL, especially for MEM workloads and when the number of threads increases. This is because when the pressure on resources is high it is more preferable to flush delinquent threads and hence free resources, than to stall these threads which causes them to hold resources for a long time. As stated before, no policy outperforms all others neither for all workloads nor for all metrics. Each one behaves better than others depending on the particular metric and workload.

VI. Improved Policies

In this section we present our improvements of the policies discussed previously.

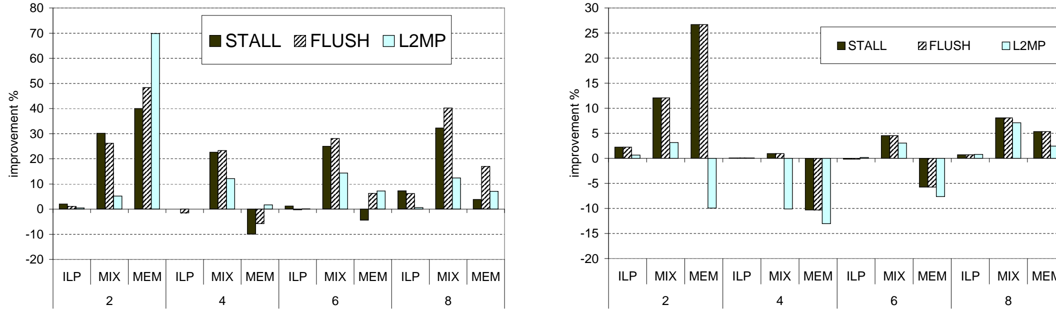


Fig. 3. Comparing current policies. (a) throughput improvement over ICOUNT; (b) Hmean improvement over ICOUNT

A. Improving L2MP

We have seen that L2MP alleviates the problem of load latency, but it does not achieve results as good as other policies addressing the same problem. The main drawback of L2MP is the AMPH loads, that is, those loads missing in the L2 cache that are not detected by the predictor. These loads can seriously damage performance because subsequent instructions occupy resources for a long time. Figure 4 indicates that this percentage is quite significant from 50% to 80%, and thus the problem still persists. We propose to add a safeguard mechanism to "filter" these undetected loads. That is, a mechanism that acts on loads missing in L2 that are not detected by the predictor. The objective is to reduce the harmful effects caused by these loads. In this paper, we use STALL [11] as a safeguard mechanism.

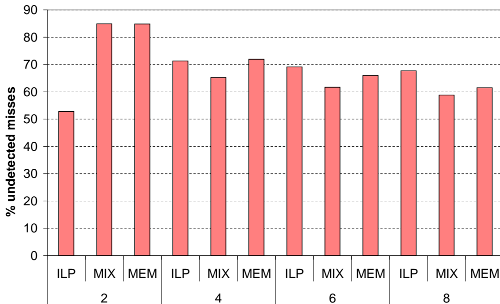
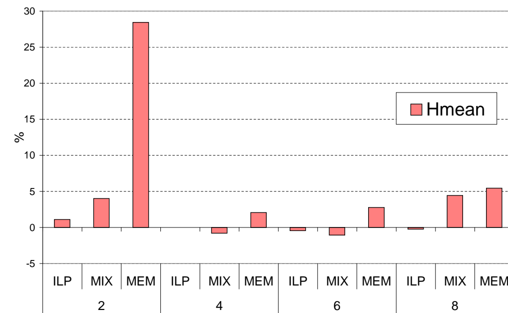


Fig. 4. Undetected L2 misses when the pattern predictor is used

Our results show that, when using L2MP, the fetch is totally idle for many cycles (15% for the 2-MEM workload) because all threads are stalled by the L2MP mechanism. Another important modification that we have made to the original L2MP mechanism, is to always keep one thread running in order to avoid idle cycles of the processor. We call the resulting policy L2MP+.

Figures 5 (a) and (b) show the throughput and the Hmean improvement of L2MP+ over L2MP. Throughput results show that for MEM workloads L2MP outperforms L2MP+ by 5.1% on average, and for MIX L2MP+ outperforms L2MP by 16% on average. We have investigated why L2MP improves L2MP+ for MEM workloads. We detected that L2MP+ significantly improves the IPC of mcf (a thread with a high L2 miss rate), but this causes an important reduction in the IPC of the remaining threads. Given that the IPC of mcf is very low, the decrease in IPC of the remaining threads affects the overall throughput more than the improvement in IPC of mcf. Hmean results, see Figure 5 (b), confirm that the L2MP+ policy presents a better throughput/fairness balance than the L2MP policy, only suffering slowdowns less than 1%.

(a) Throughput results



(b) Hmean results

Fig. 5. Improvement of the L2MP+ policy over the L2MP policy

B. Improving FLUSH

The FLUSH policy always attempts to leave one thread running. In doing so, it does not flush and fetch-stall a thread if all remaining threads are already fetch-stalled. Figure 6 shows a timing example for 2 threads. In cycle c0, thread T0 experiences an L2 miss and is flushed and fetch-stalled. After that, in cycle c1, thread T1 also experiences an L2 miss, but it is not stalled because it is the only thread running. The main problem of this policy is that by the time the missing load of T0 is resolved in cycle c2 and this thread can proceed, the machine is presumably filled with instructions from thread T1. These instructions occupy resources until the missing load of T1 is resolved in cycle c3. Hence, performance is degraded.

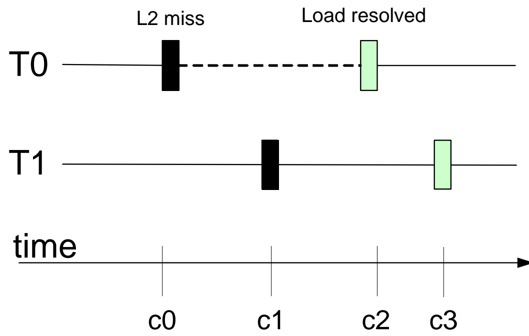


Fig. 6. Timing of the FLUSH policy

The improvement we propose is called Continue the Oldest Thread (COT). When there are N threads, $N-1$ of them are already stalled, and the only thread running experiences an L2 miss, it is stalled and flushed, but the thread that was first stalled is continued. For the previous example, the new timing is depicted in Figure 7. When thread T1 experiences an L2 miss, it is flushed and stalled and T0 is continued. Hence, instructions from T0 consume resources until cycle c2 when the missing load is resolved. However, this does not affect thread T1 because it is stalled until cycle c3. In this example, the COT improvement has been applied to FLUSH, but it can be applied to any fetch-gating policy. We have applied it also to STALL. We call the new versions of FLUSH and STALL, FLUSH+ and STALL+, respectively.

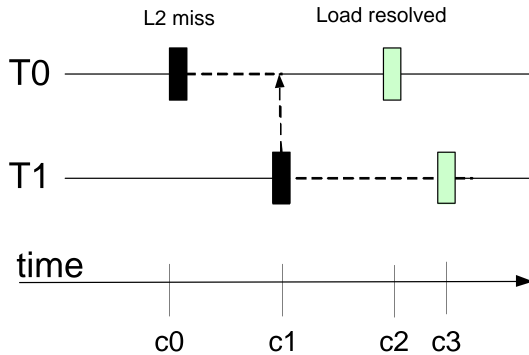
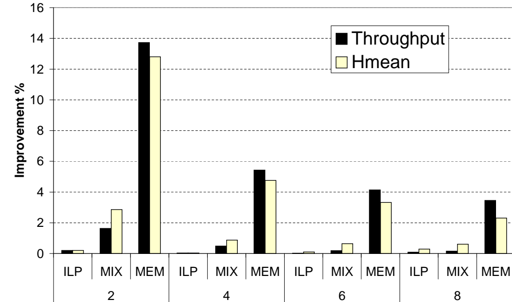
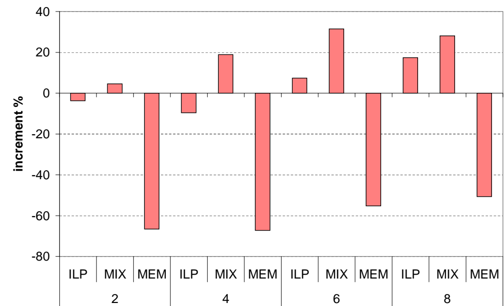


Fig. 7. Timing of the improved FLUSH policy

Figure 8 (a) shows the throughput and the Hmean improvements of FLUSH+ over FLUSH. We observe that FLUSH+ improves FLUSH for all workloads. We also observe that for MEM workloads FLUSH+ clearly outperforms FLUSH, for both metrics, and this improvement decreases as the number of threads increases. This is because, as the number of threads increases, the number of times that only one thread is running and the remaining are stopped, is lower. For MIX and ILP workloads, the improvement is lower than for the MEM workloads because the previous situation is less frequent. Concerning flushed instructions, in Figure 8(b) we see the EF improvement of FLUSH+ over FLUSH (remember the lower the value the better the result). We observe that, on average, FLUSH+ reduces EF by 60% for MEM workloads compared to FLUSH, and only increases EF by 20% for MIX workloads. These results indicate that FLUSH+ presents a better throughput/fairness balance than FLUSH, and moreover reduces extra fetch.



(a) Throughput and Hmean results



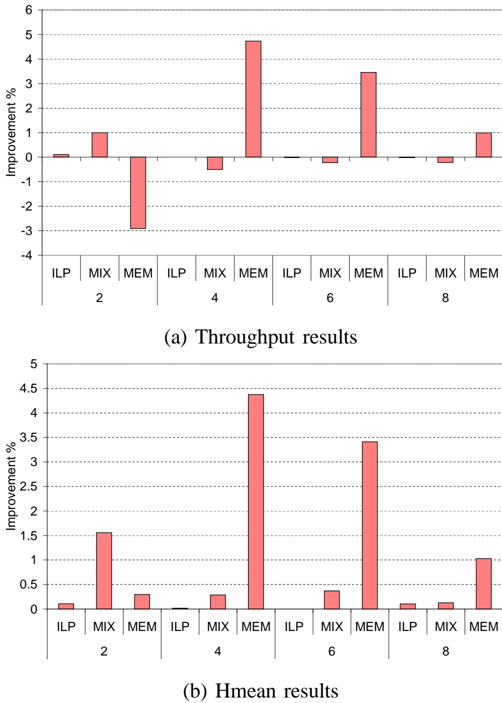
(b) EF results

Fig. 8. Improvement of the FLUSH+ policy over the FLUSH policy.

C. Improving STALL

Figures 9 (a) and (b) show the throughput and the Hmean improvement of STALL+ over STALL. We observe that the improvements of STALL+ over STALL are less pronounced than the improvements of FLUSH+ over FLUSH. Throughput results show that in general STALL+ improves STALL, and only for the 2-MEM workload there is a remarkable slowdown of 3%. The Hmean results show that STALL+ outperforms STALL for all workloads, and especially for MEM workloads. EF results do not exist because the STALL and STALL+ policies do not squash instructions.

We have analyzed why STALL outperforms STALL+ for the 2-MEM workload. We observed that this is caused by the benchmark mcf. The main characteristic of this benchmark is its high L2 miss rate. On average, one of every eight instructions is a load failing in L2. In this case, the COT improvement behaves as show in Figure 10: in cycle c0 the thread T0 (mcf) experiences an L2 miss and it is fetch-stalled. After that, in cycle c1, T1 experiences an L2 miss it is stalled and T0 (mcf) is continued. A few cycles after that, mcf experiences another L2 miss, thus control is returned to thread T1. With FLUSH, every time a thread is stalled it is also flushed. With STALL, this is not the case. Hence, from cycle c1 to c2, mcf allocates resources that are not freed for a long time, degrading the performance of T1. That is, the COT improvement for the STALL policy improves the IPC of benchmarks with high L2 miss rate, mcf in this case, but hurts the IPC of the remaining threads. This situation is especially



(a) Throughput results

(b) Hmean results

Fig. 9. Improvement of the STALL+ policy over the STALL policy. From left to right: (a) throughput results.

acute for 2-MEM workloads. To solve this problem, and other ones, we develop a new policy called FLUSH++.

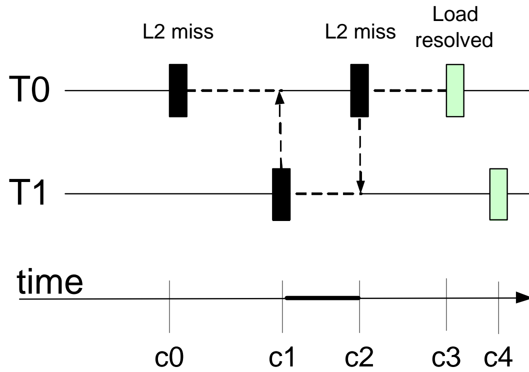


Fig. 10. Timing when a thread with high L2 miss rate is executed

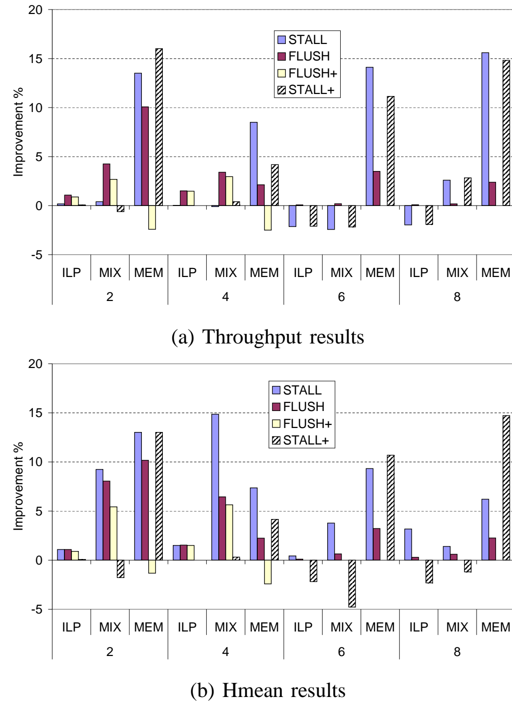
D. The FLUSH++ Policy

The FLUSH++ policy tries to combine the advantages of both previous policies, STALL+ and FLUSH+. It focuses in the following three points. First, for MIX workloads STALL+ presents good results. It is an alternative to FLUSH+ avoiding instruction re-execution. Second, another objective is to improve the IPC of STALL+ for MEM workloads with a moderate increment in the re-executed instructions. Third, the processor knows every cycle the number of threads that are running. This information can be easily obtained for any policy at runtime.

FLUSH++ works differently depending on the number of running threads. If the number of running threads is less than four, it combines STALL+ and FLUSH+. It behaves like STALL+ but when the COT improvement is triggered it acts as FLUSH+. That is, the flush is only activated when there is only one thread running and it experiences an L2 miss. In the remaining situations, threads are only stalled. When there are more than four threads running, we must consider two factors. On the one hand, the pressure on resources is high. In this situation is preferable to flush delinquent threads instead of stalling them because freed resources are highly profitable for the other threads. On the other hand, FLUSH+ improves FLUSH in both throughput and fairness for four-or-more thread workloads. For this reason, if the number of threads is greater than four, we use the FLUSH+ policy.

In Figure 11 we compare FLUSH++ with the original STALL and FLUSH policies, as well as with the improved versions STALL+ and FLUSH+. Figure 11(a) shows the throughput results and Figure 11(b) the Hmean results. We observe that FLUSH++ outperforms FLUSH in all cases in throughput as well as in Hmean. Furthermore, in Figure 12 it can been seen that for 2-, and 4-thread workloads FLUSH++ clearly reduces the EF. Concerning STALL, throughput results show that FLUSH++ only suffers a slight slowdown less than 3% for the 6-MIX workload. Hmean results show that FLUSH++ always outperforms STALL.

For ILP and MIX workloads FLUSH++ outperforms FLUSH+ and for MEM workloads it is slightly worse. The most interesting point is that, as we can see in Figure 12, FLUSH++ considerably reduces the EF of FLUSH+. For 6-, and 8-thread workloads the results are the same as those for FLUSH+.



(a) Throughput results

(b) Hmean results

Fig. 11. Improvement of the FLUSH++ policy over the FLUSH, STALL, FLUSH+ and STALL+ policies.

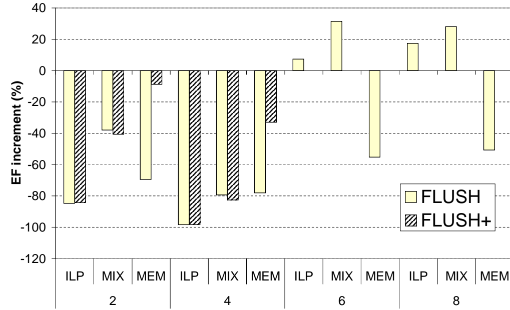


Fig. 12. EF increment of FLUSH++ over FLUSH and FLUSH+

VII. Comparing the Improved Policies

In the previous section we saw that FLUSH++ outperforms FLUSH+ and STALL+. In this section we compare FLUSH++ with L2MP+.

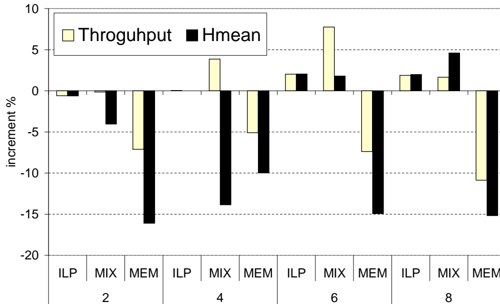


Fig. 13. Improved policies. Throughput and Hmean improvements of L2MP+ over FLUSH++

Figure 13 depicts the throughput and Hmean improvements of L2MP+ over FLUSH++. The throughput results show that L2MP+ improves FLUSH++ for MIX workloads, and that FLUSH++ is better than L2MP+ for MEM workloads. The Hmean results indicate that only for 6-, and 8-thread workloads L2MP+ is slightly more fair than FLUSH++ for ILP and MIX workloads. For the remaining workloads FLUSH++ is more fair. In general, FLUSH++ outperforms L2MP+.

VIII. Conclusions

SMT performance directly depends on how the allocation of shared resources is done. The instruction fetch mechanism dynamically determines how this allocation is carried out. To achieve high performance, it must avoid that any thread monopolizes a shared resource. An example of this situation occurs when a load misses in the L2 cache. Current instruction fetch policies focus on this problem and achieve significant performance improvements over ICOUNT.

The first contribution of this paper is that we compare three different policies addressing this problem. We show that none of the presented policies clearly outperforms all others for all metrics. The results vary depending on the particular workload and the particular metric (throughput, fairness, energy consumption, etc).

The main contributions of this paper are two. First, we analyze the use of a load miss predictor in a SMT processor. And second, we present improved versions of the three best policies addressing the described problem that have been published. Our results show that these enhanced versions achieve a significant improvements over the original ones:

- Throughput results indicate that L2MP+ outperforms L2MP for MIX workload by 16% on average and is worse than L2MP only for 2-, 4- and 6-MEM workloads (8% on average). Hmean results show that L2MP+ outperforms L2MP especially for 2-thread workloads.
- The FLUSH+ policy outperforms FLUSH in both throughput and fairness, especially for MEM workloads. Furthermore, it reduces EF by 60% for MEM workloads and only increases EF by 20% for MIX workloads.
- Throughput results show that in general STALL+ improves STALL, and only for the 2-MEM workload there is a remarkable slowdown of 3%. The Hmean results show that STALL+ outperforms STALL for all workloads, and especially for MEM workloads.
- FLUSH++, a new dynamic control mechanism, is presented. It adapts its behavior to the dynamic number of live "threads" available to the fetch logic. Due to this additional level of adaptability, it is remarkable that FLUSH++ policy fully outperforms FLUSH policy in both throughput and fairness. FLUSH++ also reduces EF for the 2- and 4-thread workloads, and moderately increases EF for the 6-MIX and 8-MIX workloads. FLUSH++ outperforms the STALL+ policy, with just a small degradation in throughput in the 6-MIX workload.

Acknowledgments

This work was 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 and Peter Knijnenburg for their comments and work in the simulation tool. The authors also would like to the reviewers for their valuable comments.

References

[1] M. Gulati and N. Bagherzadeh, "Performance study of a multithreaded superscalar microprocessor," *Proceedings of the 2nd Intl. Conference on High Performance Computer Architecture*, pp. 291–301, Feb. 1996.

[2] 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," *Proceedings of the 23th Annual Intl. Symposium on Computer Architecture*, pp. 191–202, Apr. 1996.

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

[4] W. Yamamoto and M. Nemirovsky, "Increasing superscalar performance through multistreaming," *Proceedings of the 1st Intl. Conference on High Performance Computer Architecture*, pp. 49–58, June 1995.

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

[6] J. Burns and J.-L. Gaudiot, "SMT layout overhead and scalability," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 1, pp. 142–155, Feb. 2002.

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

[8] 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*, vol. 6, no. 1, Feb 2002.

[9] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation techniques for improving load related instruction scheduling," *Proceedings of the 26th Annual Intl. Symposium on Computer Architecture*, May 1999.
[10] C. Limousin, J. Sbot, A. Vartanian, and N. Drach-Temam, "Improving 3D geometry transformations on a simultaneous multithreaded SIMD processor," *Proceedings of the 13th Intl. Conference on Supercomputing*, pp. 236–245, May 2001.
[11] D. Tullsen and J. Brown, "Handling long-latency loads in a simultaneous multithreaded processor," *Proceedings of the 34th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, Dec. 2001.
[12] A. El-Moursy and D. Albonesi, "Front-end policies for improved issue efficiency in SMT processors," *Proceedings of the 9th Intl. Conference on High Performance Computer Architecture*, Feb. 2003.
[13] K. Luo, J. Gummaraju, and M. Franklin, "Balancing throughput and fairness in SMT processors," *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pp. 164–171, Nov. 2001.
[14] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," *Proceedings of the 10th Intl. Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

Francisco J. Cazorla obtained the BS degree in Computer Science from the University of Las Palmas de Gran Canaria, Spain, in 1999, and the MS degree in Computer Science from the same University in 2001.

He is currently a Ph. D. candidate of the Polytechnic University of Catalonia. His research area focuses in instruction fetch policies for SMT architectures.

Enrique Fernández obtained his Industrial Engineering Degree from the Polytechnic University of Las Palmas in 1983 and his Ph.D in Computer Science. from the University of Las Palmas de Gran Canaria (ULPGC) in 1999. He is an assistant professor in the Informática y Sistemas Department at ULPGC. His current research interests are in the field of high performance architectures.

Alex Ramirez obtained his Computer Science degree from the Polytechnic University of Catalonia (UPC) in 1997 and his Ph.D. from the same University in 2002. Research areas of special interest are profile-guided compiler optimizations, code layout optimizations, performance studies of user and system code like database applications, and the design and implementation of the fetch stage of superscalar and multithreaded processors. He has been a student intern at Compaq's Western Research Lab. (Palo Alto, CA) and Intel's Microprocessor Research Lab. (Santa Clara, CA). Since 2000 he has been lecturing on operating systems and operating systems administration as an assistant professor. Currently Alex is involved in research and development projects with Intel and IBM.

Mateo Valero (Fellow, IEEE) obtained his Telecommunication Engineering Degree from the Polytechnic University of Madrid in 1974 and his Ph.D. from the Polytechnic University of Catalonia (UPC) in 1980. He is a professor in the Computer Architecture Department at UPC. His current research interests are in the field of high performance architectures. He has published approximately 250 papers on these topics. He served as the general chair for several conferences, including PACT-01, ISCA-98 and ICS-95, and has been an associate editor for IEEE Transactions on Parallel and Distributed Systems for three years. Dr. Valero has been honored with several awards, including the Narcis Monturiol, presented by the Catalan Government, the Salvà i Campillo presented by the Telecommunications Engineer Association and ACM, the King Jaime I by the Generalitat Valenciana, and the spanish national award "Julio Rey Pastor" for his research on IT technologies. Since 1994, he has been a member of the Royal Spanish Engineering Academy. In 2001, he was appointed Fellow of the IEEE, and since 2003 he is a Fellow of the ACM.