Improving Memory Latency Aware Fetch Policies for SMT Processors

Francisco J. Cazorla[1], Enrique Fernandez[2], Alex Ramírez[1], and Mateo Valero[1]

[1] Dpto. de Arquitectura de Computadores, Universidad Politécnica de Cataluña,
{fcazorla,aramirez,mateo}@ac.upc.es
[2] Dpto. de Informática y Sistemas, Universidad de Las Palmas de Gran Canaria,
efernandez@dis.ulpgc.es

Abstract. In SMT processors several threads run simultaneously to increase available ILP, sharing but competing for resources. The instruction fetch policy plays a key role, determining how shared resources are allocated.
When a thread experiences an L2 miss, critical resources can be monopolized for a long time choking the execution of the remaining threads. A primary task of the instruction fetch policy is to prevent this situation.
In this paper we propose novel improved versions of the three best published policies addressing this problem. Our policies significantly enhance the original ones in throughput, and fairness, also reducing the energy consumption.

Keywords: SMT, multithreading, fetch policy, long latency loads, load miss predictors

1 Introduction

Multithreaded and Simultaneous Multithreaded Processors (SMT) [3], [8], [9], [10] concurrently run several threads in order to increase available parallelism. The sources of this parallelism come from the instruction level parallelism (ILP) of each thread alone, from the additional parallelism that provides the freedom of fetching instructions from different independents threads, and from mixing them in appropriate way to the processor core. Problems arise because shared resources have to be dynamically allocated between these threads. The responsibility of the fetch policy will be 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, following instructions spend 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 represent a problem, because the ROB is not shared and the FUs are pipelined. The issue queues and the physical registers are the actual problems, because they are used

A. Veidenbaum et al. (Eds.): ISHPC 2003, LNCS 2858, pp. 70–85, 2003.
© Springer-Verlag Berlin Heidelberg 2003

for a variable, 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 [11]. 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 [4] the authors propose two mechanisms to reduce load latency: data pre-fetch and a policy based on a load miss predictor (we will explain these policies in the related work section). STALL [7], fetch-stalls a thread when it is declared to have an L2 missing load until the load is resolved. FLUSH [7] works similarly and additionally flushes the thread which the missing load belongs to. DG [2] and PDG [2] are two recently proposed policies that try to reduce the effects of L1 missing loads. Our performance results show that FLUSH outperforms both policies, hence we will not evaluate DG and PDG in this paper.

In the first part of this paper we analyze the space of parameters of STALL, FLUSH and a policy based on the usage of load miss predictors (L2MP), and compare their effectiveness. As we will see none of them outperforms all other in all cases, but each behaves better depending on the metric and on the workload. Based on this initial study, in the second part we propose improved versions of each of them. Throughput and fairness [5] results show that in general improved versions achieve important performance increments 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 explain the current policies. Section 5 compares the effectiveness of those policies. In Section 6 we propose several improvements for the presented polices. Section 7 compares the improved policies. Finally Section 8 is devoted to the conclusions.

2 Related Work

Current I-fetch policies address the problem of L2 missing loads latency in several ways. Round Robin [8] is absolutely blind to this problem. Instructions are alternatively fetched from available threads, even when any of them has in-flight L2 misses. ICOUNT [8] only takes into account the occupancy of the issue queues, and disregards that a thread can be blocked on an L2 miss, while making 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. On the contrary, 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 [11] a load hit/miss predictor is used in a super-scalar processor to guide the dispatch of instructions made by the scheduler. This allows the scheduler to dispatch dependent instructions exactly when the data is available. 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 fetch, switching between threads when any of them is predicted to miss in L2.

In [4] the authors propose two mechanisms oriented to reduce the problem associated to load latency. 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 the whole resources.

In [7] 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 mechanism present good results: the first one, STALL, consists of fetch-stalling the offending thread. The second one, FLUSH, flushes the instructions after the L2 missing load, and also it 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 results show performance improvements over ICOUNT for some workloads, especially for workloads with a few number of threads. However, FLUSH requires complex hardware, and increase 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. However, in general its performance results are worse.

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

3 Metrics and Experimental Setup

We have used three different metrics to make a fair comparison of the policies: the IPC throughput, a metric that balances throughput and fairness (Hmean), and a metric that takes into account the extra energy used due the re-execution 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 [5]. Hmean is calculated as shown in Formula 1.

$$Hmean = \frac{\#threads}{\sum_{threads} \frac{IPC_{alone}}{IPC_{wld}}} \ . \tag{1}$$

The extra fetch (EF) metric measures the extra instructions fetched due to the flush of instructions (see Formula 2). Here we are not taking into account the flushed instructions due to branch mispredictions, but only those related with the loads missing in L2. EF compares the total fetched instructions (flushed and not flushed) with the instructions that are fetched and not flushed. The higher the value of the EF the higher the number of squashed instructions respect to the total fetched instructions. If no instructions is squashed, EF is equal to zero because the number of fetched instructions is equal to the number of fetched and not squashed instructions.

$$EF = \frac{TotalFetched * 100}{Fetched\ and\ not\ squashed} - 100\ (\%) \ . \tag{2}$$

We have used a trace driven SMT simulator, based on SMTSIM [9]. It consists of our own trace driven front-end and a modified version of SMTSIM's back-end. Baseline configuration is shown in Table 1 (a).

Traces were collected of the most representative 300 million instruction segment following the idea presented in [6]. The workload consists on 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 DEC Alpha AXP-21264 C/C++ compiler. Programs are divided in two groups based on their cache behavior (see Table 1 (b)): those with an L2 cache miss rate higher than 1%[1] are considered memory bounded (MEM), the rest are considered ILP. From these programs we create 12 workloads, as shown in Table 2, ranging 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 were used twice, because there are not enough SPECINT benchmarks with bad cache behavior. The replicated benchmarks are boldfaced in Table 2. We have shifted second instances of replicated benchmarks by one million instructions in order to avoid both threads accessing the cache hierarchy at the same time.

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

Table 1. 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 2. 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**

4 Current Policies

In this section we discuss several important issues about the implementation of the policies that we are going to evaluate: L2MP, STALL and FLUSH.

In this paper we evaluate a policy that uses predictors to predict L2 misses. We call this policy L2MP. The L2MP mechanism is shown in Figure 1. The predictor acts in the decode stage. It is indexed with the PC of the loads: 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 is continued.

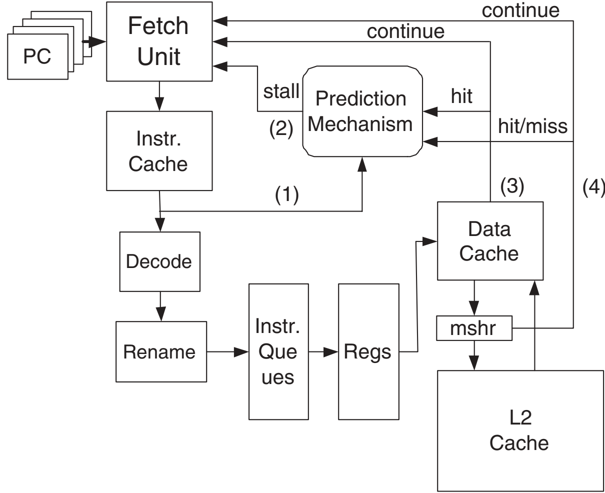

Fig. 1. L2MP mechanism

We have explored a wide range of different load miss predictors [1]. The one that obtains the best results is the predictor proposed in [4], what we call predictor of patterns.

About FLUSH and STALL, in [7] a load is declared to miss in L2 when it spends more than 15 cycles in the memory hierarchy. We have experiment different values for this parameter, and 15 presents the best overall results for our baseline architecture. Three additional considerations about FLUSH and STALL are: a data TLB miss also triggers a flush (or stall); a 2-cycle advance indication is received when a load returns from memory; and 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.

5 Comparing the Current Policies

In this section we will determine the effectiveness of the different policies addressing the problem of load latency. We will compare the STALL, FLUSH and L2MP policies using the throughput and the Hmean.

In Figure 2 we show the throughput and the Hmean improvements of STALL, FLUSH, and L2MP over ICOUNT. L2MP achieves important throughput increments over ICOUNT, mainly for 2-thread workloads. However, fairness results using the Hmean metric indicates 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 it is because L2MP hurts MEM threads and boosts ILP threads, especially for few-thread workloads.

If we compare the effectiveness of L2MP with other policies addressing the same problem, like STALL, we observe that L2MP only achieves better through-

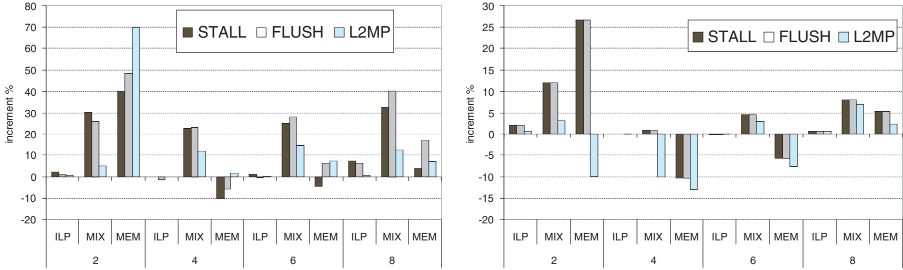

Fig. 2. Comparing current policies. (a) throughput increment over ICOUNT; (b) Hmean increment over ICOUNT

put than STALL for MEM workloads. However, L2MP heavily affects fairness. We will explain why L2MP does not obtain results as good as STALL soon.

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 preferable to flush delinquent threads, and hence free resources, than stall these threads holding resources for a long time.

As stated before, no policy outperforms the other neither for all workloads, nor for all metrics. Each one behaves better depending on the particular metric and workload.

6 Improved Policies

6.1 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 are the loads missing in L2 cache that are not detected by the predictor. These loads can seriously damage performance because following instructions occupy resources for a long time. Figure 3 depicts the percentage of missing loads that are not detected by the predictor of patterns. This percentage is quite significative (from 50% to 80%), and thus the problem still persists.

We propose to add a safeguard mechanism to “filter” this undetected loads. That is, this mechanism acts on loads missing on L2 that are not detected by the predictor. The objective is to reduce the harmful effects caused by these loads. In this paper we have used STALL [7] as safeguard mechanism.

Our results show that, when using L2MP, the fetch is absolutely idle for many cycles (i.e. 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 maintain always one thread running in order to avoid idle cycles of the processor.

The Figures 4 (a) and (b) show the throughput and the Hmean increment of L2MP+ over L2MP. Throughput results show that for MEM workloads L2MP

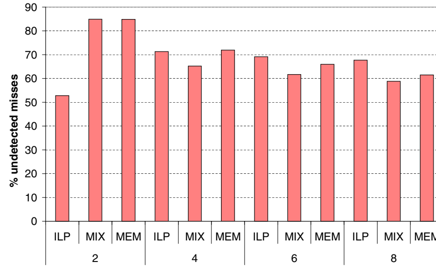

Fig. 3. Undetected L2 misses

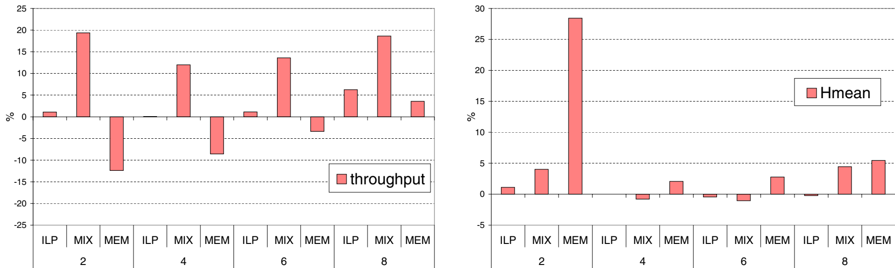

Fig. 4. L2MP+ vs. L2MP. (a) throughput results; (b) Hmean results

outperforms L2MP+ (5.1% on average), and for MIX L2MP+ outperforms L2MP (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. And given that the IPC of *mcf* is very low, the decrement in the IPC of the remaining threads affects more the overall throughput than the increment in the IPC of *mcf*. Table 3 shows the relative IPC of *mcf* and the remaining threads for each MEM workload. In all cases L2MP+ improves the IPC of *mcf* and degrades the IPC of the remaining threads. This indicates that the

Table 3. Relative IPCs

RELATIVE IPCs		L2MP	L2MP+	Increment
2 – MEM	mcf	0.32	0.64	100
	remaining	0.72	0.60	-17.01
4 – MEM	mcf	0.30	0.48	61.36
	remaining	0.39	0.35	-10.07
6 – MEM	mcf	0.29	0.39	33.72
	remaining	0.30	0.27	-7.63
8 – MEM	mcf	0.28	0.34	21.69
	remaining	0.19	0.19	0.48

original policy favors ILP threads but it is at the cost of hurting MEM threads. Hmean results, see Figure 4 (b), confirm that the L2MP+ policy presents a better throughput-fairness balance than the L2MP policy. L2MP+ only suffers slowdowns lower than 1%.

6.2 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. The Figure 5 shows a timing example for 2 threads. In the cycle c0 the thread T0 experiences an L2 miss and it 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 (cycle c2), and this thread can proceed, the machine is presumably filled with instructions of thread T1. These instructions occupy resources until the missing load of T1 is resolved in cycle c3. Hence, performance is degraded.

The improvement we propose is called Continue the Oldest Thread (COT), and it is the following: when there are N threads, N-1 of them are already stalled, and the only thread running experiences an L2 miss, it is effectively stalled and flushed, but the thread that was first stalled is continued. In the previous example the new timing is depicted in Figure 6. When thread T1 experiences an L2 miss it is flushed and stalled, and T0 is continued. Hence, instructions of T0 consume resources until cycle c2 when the missing load is resolved. However, this does not affect to the thread T1 because it is stalled until cycle c3. In this example COT improvement has been applied to FLUSH, but it can be applied to any of the fetch-gating policies. We have applied it also to STALL. We call the new versions of FLUSH and STALL, FLUSH+ and STALL+.

Figure 7 (a) shows the throughput and the Hmean increments of FLUSH+ over FLUSH. We observe that FLUSH+ improves FLUSH for all workloads. We

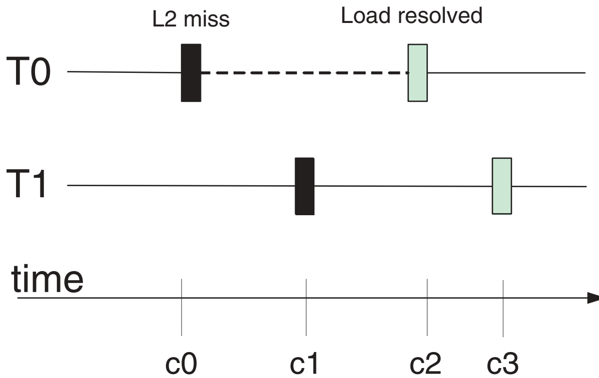

Fig. 5. Timing of the FLUSH policy

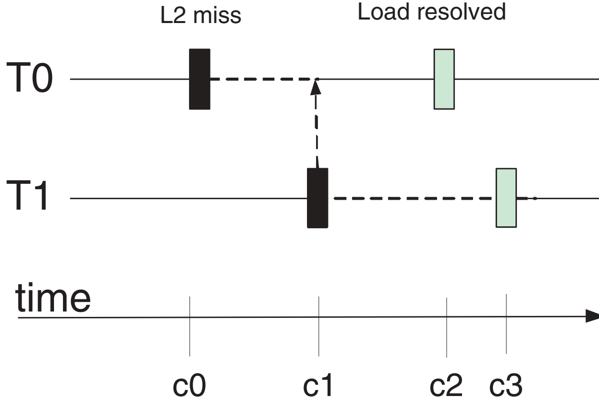

Fig. 6. Timing of the improved FLUSH policy

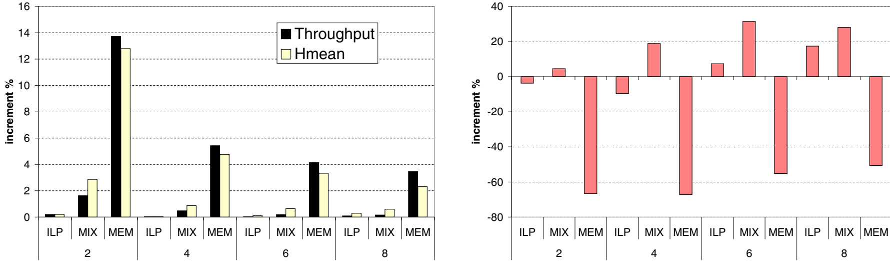

Fig. 7. FLUSH+ vs. FLUSH. (a) throughput and Hmean results; (b) EF results

also observe that for MEM workloads FLUSH+ clearly outperforms FLUSH, for both metrics, and the improvement decreases as the number of thread increases. This is because as the number of threads increases the number of time 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 also less frequent. Concerning to flushed instructions, in Figure 7(b) we see the EF increment of FLUSH+ over FLUSH (remember the lower the value the better the result). We observe that, on average, FLUSH+ decrements by 60% FLUSH for MEM workloads and only increments by 20% FLUSH for MIX workloads. These results effectively indicate that FLUSH+ presents a better throughput-fairness balance than FLUSH, and also reduces extra fetch.

6.3 Improving STALL

The improved STALL policy, or STALL+, consists of applying the COT improvement to STALL.

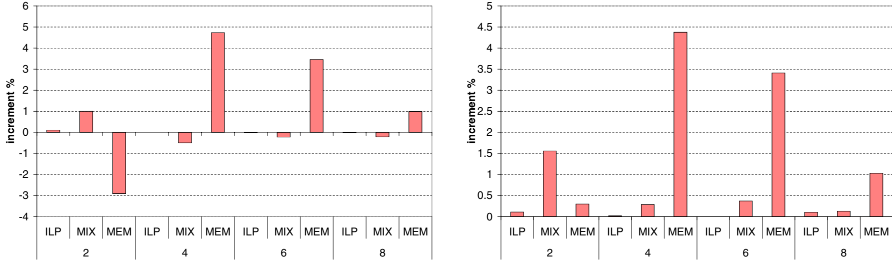

Fig. 8. STALL+ vs. STALL. (a) throughput results; (b) Hmean results

Figures 8 (a) and (b) show the throughput and the Hmean increment of STALL+ over STALL. We observe that the improvements of STALL+ over STALL are less pronunced 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. The EF results are not shown because the STALL and STALL+ policies do not squash instructions.

We analyzed why STALL outperforms STALL+ for the 2-MEM workload. We observe that the cause is 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 9: 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. Few cycles after that, *mcf* experiences another L2 miss, thus the control is returned to thread T1. The point is that 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, COT improvement for STALL policy improves the IPC of benchmarks with high L2 miss rate, *mcf* in this case, but it hurts the IPC of the remaining. This situation is especially acute for 2-MEM workloads. To solve this problem, and other ones, we develop a new policy called FLUSH++.

6.4 FLUSH++ Policy

This policy tries to obtain the advantages of both policies, STALL+ and FLUSH+, and it focuses in the following points:

- For MIX workloads STALL+ presents good results. It is an alternative to FLUSH+ avoiding instruction re-execution.
- Another objective is to improve the IPC of STALL+ for MEM workloads with a moderate increment in the re-executed instructions.

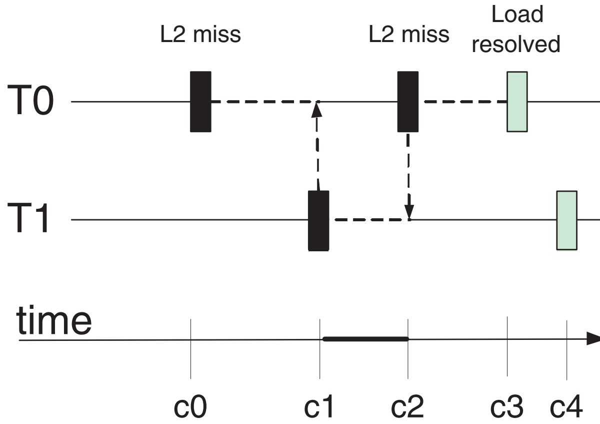

Fig. 9. Timing when the *mcf* benchmark is executed

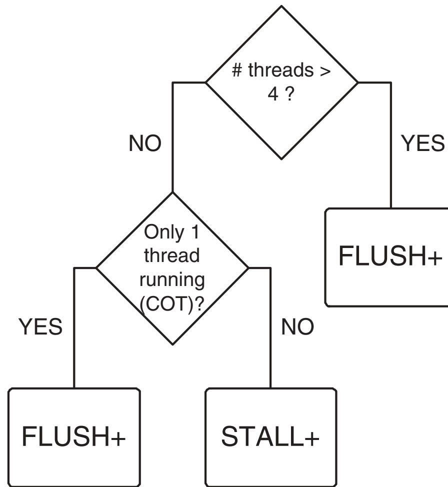

Fig. 10. FLUSH++ policy

– 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, see Figure 10.

– If the number of running threads is less than four, it combines STALL+ and FLUSH+. In a normal situation 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.

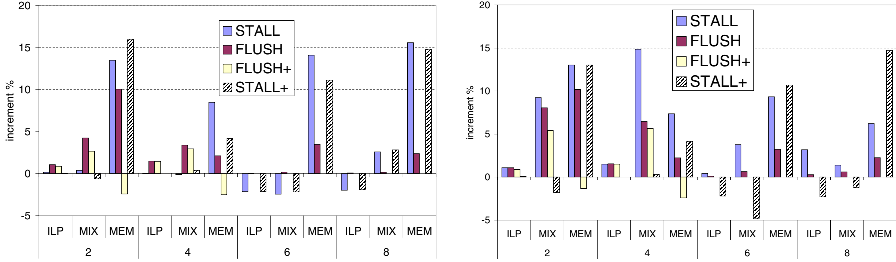

Fig. 11. FLUSH++ vs. FLUSH, STALL, FLUSH+ and STALL+. (a) throughput results; (b) Hmean results

– 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 it because freed resources are highly profited by 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 will 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+. The Figure (a) shows the throughput results, and the Figure (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 slight slowdowns lower than 3% for the 6-MIX workload. Hmean results show that FLUSH++ always outperforms STALL.

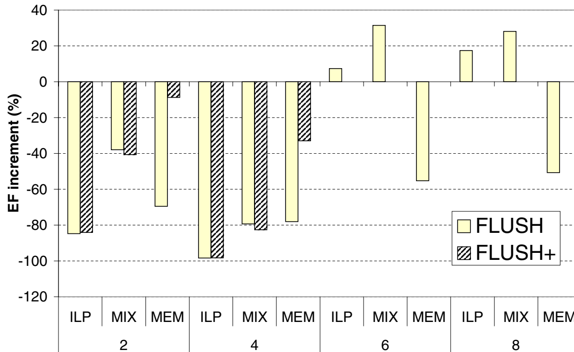

Fig. 12. FLUSH++ increments in EF over FLUSH, and FLUSH+

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 the Figure 12 FLUSH++ considerably reduces the EF of FLUSH+. For 6-, and 8-thread workloads the results are the same that for FLUSH+.

7 Comparing the Improved Policies

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

Figure 13 depicts the throughput and Hmean increments 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+, however for some configurations this is not the case. This confirms that each policy presents better results than the remaining depending on the particular workload and metric.

8 Conclusions

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

A minor contribution of this paper is that we compare three different policies addressing this problem. We show that none of the presented policies clearly out-

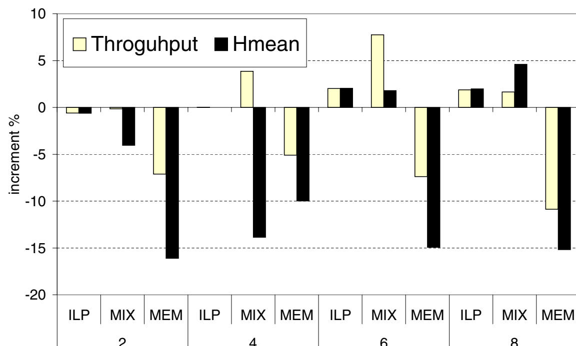

Fig. 13. Improved policies. Throughput and Hmean increments of L2MP+ over FLUSH++

performs all others for all metrics. The results vary depending on the particular workload and the particular metric (throughput, fairness, energy consumption, etc). The main contribution is that we have presented four improved versions of the three best policies addressing the described problem. Our results show that this enhanced versions achieve a significant improvement over the original ones:

- The throughput results indicate that L2MP+ outperforms L2MP for MIX workload (16% on average) and is worse than L2MP only for 2-, 4- and 6-MEM workloads (8% on average). The 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 extra fetch by 60% for MEM workloads and only increments extra fetch 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 "alive 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. Regarding STALL+, FLUSH++ outperforms STALL+ policy, with just a slight exception in throughput in 6-MIX workload.

Acknowledgments

This work was supported by the Ministry of Science and Technology of Spain under contract TIC-2001-0995-C02-01, and grant FP-2001-2653 (F. J. Cazorla), and by CEPBA. The authors would like to thank Oliverio J. Santana, Ayose Falcon and Fernando Latorre for their comments and work in the simulation tool. The authors also would like to the reviewers for their valuable comments.

References

1. F.J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Improving long-latency-loads-aware fetch policies for SMT processors. Technical Report UPC-DAC-2003-21, Universitat Politecnica de Catalunya, May 2003.
2. A. El-Moursy and D.H. Albonesi. Front-end policies for improved issue efficiency in SMT processors. *Proceedings of the 9th Intl. Conference on High Performance Computer Architecture*, February 2003.
3. M. Gulati and N. Bagherzadeh. Performance study of a multithreaded superscalar microprocessor. *Proceedings of the 2nd Intl. Conference on High Performance Computer Architecture*, pages 291–301, February 1996.

4. C. Limousin, J. Sébot, A. Vartanian, and N. Drach-Temam. Improving 3d geometry transformations on a simultaneous multithreaded SIMD processor. *Proceedings of the 13th Intl. Conference onSupercomputing*, pages 236–245, May 2001.
5. 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*, pages 164–171, November 2001.
6. T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. *Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, September 2001.
7. 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*, December 2001.
8. 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*, pages 191–202, April 1996.
9. D.M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *Proceedings of the 22th Annual Intl. Symposium on Computer Architecture*, 1995.
10. W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. *Proceedings of the 1st Intl. Conference on High Performance Computer Architecture*, pages 49–58, June 1995.
11. 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.