37th International Conference on Parallel Processing

MFLUSH: Handling Long-latency loads in SMT On-Chip Multiprocessors

Carmelo Acosta †, Francisco J. Cazorla ⋆, Alex Ramirez †⋆, Mateo Valero †⋆

† Universitat Politecnica de Catalunya
HiPEAC European Network of Excellence
Barcelona, Spain
{cacosta,aramirez,mateo}@ac.upc.edu

⋆ Barcelona Supercomputing Center
Barcelona, Spain
{francisco.cazorla,alex.ramirez,mateo.valero}@bsc.es

Abstract

*Nowadays, there is a clear trend in industry towards employing the growing amount of transistors on chip in replicating execution cores (*CMP*), where each core is* Simultaneous Multithreading *(*SMT*). State-of-the-art high-performance processors like the IBM POWER5 and POWER6 corroborate this* CMP+SMT *trend. Within each* SMT *core any of the well-known SMT mechanisms may be applied to face SMT related challenges. Among them, probably the most important issue in an* SMT *execution pipeline concerns the* Instruction Fetch (IFetch) Policy*. The* FLUSH *IFetch Policy represents a choice for throughput-oriented scenarios. It handles L2 cache misses in order to avoid hardware resource monopolization by any given execution thread; involving an additional energy cost via instruction refetching. However, the new constraints imposed by the* CMP+SMT *scenario may affect well-known SMT mechanisms, like the* FLUSH *mechanism.*

In this paper we revisit the FLUSH *mechanism and analyze its application in the emerging* CMP+SMT *scenario. The included analysis points out the new difficulties to be faced by the* FLUSH *mechanism in the emerging* CMP+SMT *scenario. Then we propose a novel* IFetch Policy *designed to cope with the* CMP+SMT *scenario: the* MFLUSH*. We also include a complete evaluation of the* MFLUSH *policy, both in terms of throughput and energy consumption. Our results indicate that the* MFLUSH*, specifically designed for the emerging* CMP+SMT *scenario, succeeds not only in overcoming the specific* CMP+SMT *constraints but also allowing a 20% energy consumption reduction without a significant system throughput loss.*

1 Introduction

As process technology advances, and we have at our disposal more transistors on a single chip, the issue of how to effectively employ so many resources gains importance. This quest of effectiveness led to *Simultaneous Multithreading* (*SMT*) [7, 13, 16] and *On-Chip Multiprocessors* (*CMP*) [8]. Nowadays, there is a clear trend in industry towards *CMP* or even *CMP+SMT* processors, like the Intel Core 2 Duo [15], IBM POWER5 [11] and POWER6 [6], and Sun T1 [2] and T2 [1] Niagara processors. This trend seems to head towards high-degree *CMPs* [1], with lots of on-chip cores.

On the one hand, conventional *CMP* designs share the second level (L2) cache among all the on-chip cores by means of an interconnection switch. As the number of on-chip cores increases, the pressure on both the L2 cache and the interconnection network is also augmented. As a result, the L2 cache access time turns more *unpredictable*.

On the other hand, the L2 cache access time is used in *SMT* processors to detect L2 cache misses. As shown by Tullsen et al. in [12], L2 cache misses are of key importance in *SMTs*. Thus, a long latency instruction, like an L2 cache miss, in any running thread may stall the whole machine. The *Instruction Fetch (IFetch) Policy* may avoid these harmful situations, determining from which thread(s) instructions are fetched every cycle. Several authors have shown that long latency operations have to be taken into account by the *IFetch Policy* in order to boost *SMT* performance [3, 4, 12, 14]. Some of these *IFetch Policies* track the delay of loads when accessing the outer cache level (the L2 cache in our processor setup) in order to determine whether

[1]In this paper the term *degree of a CMP* refers to the number of cores of the *CMP*. Analogously, the term *degree of an SMT* refers to the number of contexts of that *SMT*. For example, the IBM POWER5 is a 2-degree *CMP* where each core is a 2-degree SMT.

0190-3918/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPP.2008.48

IEEE computer society

they miss. Once an L2 cache miss is detected the corresponding thread is stopped/flushed to prevent resource monopolization.

In this paper we shed some light on the implications of having multiple *SMT* cores sharing a single L2 cache. We focus our analysis on the application of the *FLUSH* [12] IFetch Policy to the emerging *CMP+SMT* scenario, with multiple *SMT* cores sharing an L2 cache. As we augment the number of replicated SMT cores sharing the same L2 cache both the memory traffic (between each core and L2 cache) and the contention (L2 cache banks and ports) increases, resulting in new challenges to be faced up. From this analysis, we propose a novel *IFetch Policy* designed to cope with the emerging *CMP+SMT* scenario: the *MFLUSH*. We include a complete evaluation of the *MFLUSH* both in terms of throughput and energy consumption. Our results indicate that the *MFLUSH*, specifically designed for the emerging *CMP+SMT* scenario, succeeds not only in overcoming the specific *CMP+SMT* constraints but also allowing a 20% reduction in the required energy consumption without a significative (less than 3%) system throughput loss.

2 Methodology

We use a trace driven SMT simulator derived from SMTsim [13]. The simulator consists of our own trace driven front-end and an improved version of the SMTsim's back-end that provides multicore support. Our simulator also permits simulating the impact of executing along wrong paths on the branch predictor and the instruction cache by having a separate *basic block dictionary*, in which is contained information of all static instructions.

Our workloads use the *SPEC2000 benchmark suite*. We have collected traces of the most representative 300 million instruction segment of each benchmark, following the idea presented in [9]. Each program is compiled with the *–O2 –non_shared* options using DEC Alpha AXP-21264 C/C++ compiler and executed using the reference input set. Since a complete study of all benchmarks is not feasible due to excessive simulation time, we have randomly chosen some of them comprising 5 workloads for 4 different workload sizes (i.e., 20 workloads). The name of each workload is xWy, where x stands for the number of threads involved and y stands for the workload identifier (e.g., *6W2* identifies the second workload with 6 threads). Each workload size x is simulated on a *CMP+SMT* implementation with $\frac{x}{2}$ two-hardware-context SMT cores. In order to obtain comparable results using different *IFetch Policies* all simulations are executed for a fixed interval of 120 millions of simulation cycles. Figure 1 shows the main simulation parameters and the chosen workloads.

Core Parameters	
Pipeline depth	11 stages
Queues Entries	64 int, 64 fp, 64 ld/st
Execution Units	4 int, 3 fp, 2 ld/st
Physical Registers	320 regs.
ROB Size*	256 entries
Branch Predictor	perceptron (4K local, 256 perceps.)
BTB	256 entries, 4-way associative
RAS*	100 entries

Cache Hierarchy Parameters	
L1 icache	64KB, 4-way, 8 banks
L1 dcache	32KB, 4-way, 8 banks
L1 lat./miss	3/22 cycs.
I-TLB ,D-TLB	512 ent. Full-associative
TLB miss	300 cycs.
L2 Cache	4MB, 12-way, 4 banks
L2 latency	15 cycs.
Main Memory lat.	250 cycs.

gzip	a	eon	h	apsi	o	facerec	v
vpr	b	gap	i	wupwise	p	applu	w
gcc	c	vortex	j	equake	q	galgel	x
mcf	d	bzip2	k	lucas	r	ammp	y
crafty	e	twolf	l	mesa	s	mgrid	z
perlbmk	f	art	m	fma3d	t		
parser	g	swim	n	sixtrack	u		

	Number of Threads			
Name	2	4	6	8
xW1	b, j	b, q, t, j	l, b, q, f, t, j	d, l, b, g, i, j, c, f
xW2	n, e	l, n, p, e	g, l, n, p, e, a	b, g, m, n, a, h, o, p
xW3	d, a	d, s, r, a	d, l, s, w, r, a	m, n, r, q, i, j, e, h
xW4	g, f	g, b, m, f	r, g, b, m, h, f	l, b, g, m, n, r, f, s
xW5	r, p	r, j, f, p	h, l, e, r, m, d	q, b, c, k, e, a, o, t

Figure 1. Simulation parameters and Workloads. (resources marked with * are replicated per thread)

3 Analysis

In our research we focus on *CMPs* comprised of *SMT* cores, or simply *CMP+SMT*. Each *SMT* core allows two threads running simultaneously and has its private instruction and data cache (see details in Figure 1). The first level cache is connected, through an on-chip bus-based interconnection network, to a shared multibanked L2 cache. The icache and dcache of each core is connected to all the shared L2 cache banks.

In our analysis, we evaluate the interaction between the shared L2 cache and the *IFetch Policy* implemented within each SMT core. Both the memory traffic, between L1 and L2 caches, and contention effects, regarding the use of each shared L2 cache bank, are considered. Two well-known SMT *IFetch Policies* are used in our research: *ICOUNT* and *FLUSH*.

The *ICOUNT* policy [14] prioritizes threads with fewer instructions in the pre-issue stages, and presents good results for threads with high *Instruction Level Parallelism* (*ILP*). However, *SMTs* have difficulties with threads that experience many loads that miss in the L2 cache. When this situation happens, the *ICOUNT* does not realize that a thread can be blocked on an L2 cache miss and will not make forward progress for many cycles. Depending on the amount of instructions dependent of the blocked load, many processor resources may be blocked and the total throughput suffers from a serious slowdown.

The performance of *IFetch 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 (*Non-Speculative implementation*), we know for certain that it is a delinquent load: *totally reliable* but *too late*. On the other hand, we can predict (*Speculative implementation*) which loads are going to miss by adding a load miss predictor to the front-end. In this case, the *speed* is *higher*, 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 *IFetch Policy* takes for delinquent threads. With these two parameters, we will classify all current policies related to long latency loads.

In [12] several *RA* are proposed. We focus on the mechanism leading to the best performance, called *FLUSH*. As a result of applying *FLUSH*, the offending thread temporarily does not compete for resources. More importantly, the hardware resources used by this thread are freed, giving the other threads full access to them. Several *DM* are proposed for the *FLUSH* response action.

- *Delay 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. We will refer to this *FLUSH*'s *DM* as *Speculative* (*FL-SX*), where *X* stands for the delay (cycles) after which the mechanism is triggered.

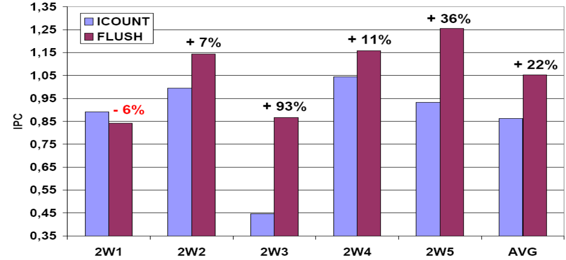


Figure 2. Throughput in single-core SMT.

- *Trigger on miss DM:* In this case we wait until the load miss in the L2 cache to start the corresponding *RA*. We will refer to this *FLUSH*'s *DM* as *Non-Speculative* (*FL-NS*).

3.1 Single-core analysis

According to our simulation parameters (see Figure 1) we chose *30 cycles* (*FL-S30*) as *FLUSH* trigger, that is the delay waited prior to activate the *FLUSH* mechanism once a load is issued from the corresponding queue.

Our results are consistent with [12]: the *delay-after-issue DM* yields better results than *trigger on miss*, both improving *ICOUNT*. For this experiment, we simulated a single-core SMT configuration. In this uniprocessor, with two hardware contexts, we ran all 2-thread (*2Wy*) workloads in Figure 1. Figure 2 shows the comparison between ICOUNT and Speculative FLUSH (FL-S30) results. From these results it can be asserted that the *FLUSH* mechanism effectively reduces system throughput losses in workloads containing threads with bad memory behaviors. Thus, the *FLUSH* mechanism yields speedups of up to 93%, with average speedup of 22%. However, as described in the following section, these asserts are highly dependent on the amount of replicated SMT cores.

3.2 Multiple-core analysis

Next, we simulated the remainder workloads in Figure 1, replicating *SMT* cores with two threads per core. Figure 3 shows the average results per each workload size. These results point out that the prior asserts made for the single-core case, regarding the performance of the *FLUSH* mechanism, are not valid for the multicore *CMP+SMT* configurations. In fact, as we increase the amount of replicated *SMT* cores the 22% average speedup, obtained with the *FLUSH* mechanism in a single-core SMT when compared to *ICOUNT*, experiences a progressive reduction. With a 4-core configuration (8 thread workloads - *8Wy*), the *FLUSH* mech-

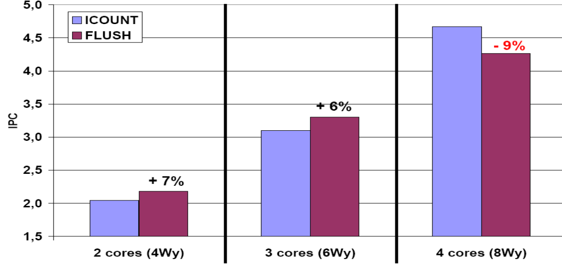


Figure 3. Average throughput in multicore CMP+SMT configurations.

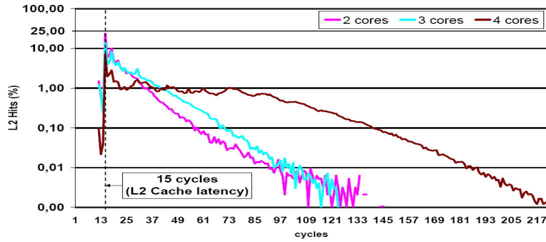


Figure 4. Average L2 cache hit time.

anism's performance improvement disappears yielding a 9% average slowdown.

In order to shed some light into the rationale behind these results, we deeply analyzed the influence of the access time to the shared L2 cache. Figure 4 shows the average number of cycles required for each load that hits on the shared L2 cache, since it is issued from the load/store queue until it is finally served. For this measurement we use the *ICOUNT* policy since it does not alter the L2 cache access patern.

Figure 4 points out that the probability of suffering from high latencies in L2 cache accesses increases with the amount of SMT cores. As indicated in Figure 1, each of the 4 banks of the shared L2 cache is single-ported and has an access latency of 15 cycles. That is, two consecutive accesses to the same L2 cache bank cannot be served in less than 15 cycles. Each *SMT* core implements 2 Load/Store Units, shared by the two threads running in the core. Within each core it is also implemented a 16-entry MSHR queue that keeps track of the outstanding memory requests. In case of L2 hits, consecutive accesses to the same L2 cache bank may overlap yielding a higher access time. As an example, the fourth consecutive L2 hit to the same L2 cache bank would experience a 45-cycle delay. Each additional SMT core increases in 2 the number of loads that can be issued in a single cycle, with the consequent increment of the pressure on both the interconnection network (L1-L2 bus) and the shared L2 cache.

Figure 4 also indicates that the dispersion of the L2 access time also increases with the number of SMT cores. Focusing on the average L2 hit time for a 4-core implementation in Figure 4, about half the L2 hits are equally distributed in the range of 20-70 cycles. This fact points out that there is no a single threshold, to be used as trigger value for the *FLUSH* mechanism, which provides good results for all cases. This high variability in the L2 cache access time hampers the predictability of the L2 behavior:

- On the one hand, if we set a low threshold value the number of *false misses* increases. That is, the number of long-latency L2 hits predicted as L2 misses. As a result, the performance of the *FLUSH* policy is heavily affected.
- On the other hand, if we set a high threshold value the number of cycles a thread can clog resources increases, leading to performance loss. We comment this issue in the next section.

To sum up, the performance of the *FLUSH* mechanism exhibits a clear trend to get diminishing returns as we increase the number of *SMT* cores in a *CMP+SMT* scenario. In fact, the *FLUSH* mechanism turns ineffective just by passing from a dual core to a quad core implementation, as depicted in Figure 3.

3.3 Detection Moment Analysis

The results in Figure 4 exhibit higher levels of dispersion as the number of *SMT* cores increases. In this section we analyze how does this issue affect the choice of the right trigger for the FLUSH mechanism. Thus, we ran some additional simulations covering a wider *DM* spectrum. For an explanatory analysis, we chose two representative 8-thread workloads: *(a) 8W3* (see Figure 1) and *(b)* an 8-thread workload comprised of instances of *bzip2* and *twolf*, where instances of the two applications never share a single core. Figure 5 shows the results obtained using different values for the FLUSH's trigger, ranging from 30 to 150 cycles. The non-speculative implementation (FL-NS) is also included.

In Figure 5*(a)*, the trigger that yields the highest throughput is 50 cycles. However, compared to speculative instances, the non-speculative FLUSH implementation yields the highest overall throughput. In Figure 5*(b)*, the best trigger value is 90 cycles. These examples illustrate that there may be different trigger values which best balance the amount of false misses and clog resources, yielding the highest overall throughput. That is, the choice of the right value depends on each specific workload.

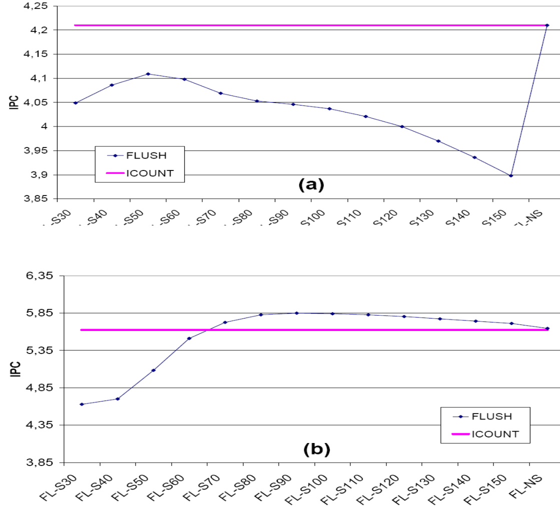


Figure 5. Detection Moment Analysis.

4 The MFLUSH Policy

The MFLUSH mechanism adapts the FLUSH [12] and STALL [12] phylosophy to the emerging CMP+SMT scenario. Built on top of ICOUNT [13], the MFLUSH mechanism avoids the waste of resources by threads blocked waiting for memory. Whenever a thread waits for a memory access to be resolved, the MFLUSH mechanism predicts its resolution time and reacts accordingly. Since the CMP+SMT scenario has less memory access predictability than the prior SMT scenario, this issue turns into a non-trivial task. The MFLUSH is designed to cope with the varying workload behavior and memory traffic conditions of the emerging CMPs comprised of SMT cores sharing one or multiple L2 Caches. Thus, it adapts its L2 miss predictions to the varying conditions instead of using an heuristic prediction value, as done in FLUSH.

The MFLUSH mechanism establishes, according to the specific system characteristis, an operational environment as shown in Figure 6. For each memory access the MFLUSH mechanism predicts its resolution time, based on prior accesses. These predictions fall in the MIN - MAX range (See Figure 6), where MIN and MAX correspond to the L1 and L2 cache miss latency, respectively. As seen in prior sections, the access time of an L2 cache may experience high variability when multiple SMT cores share it. The more cores sharing a single L2 cache and interconection bus, the more traffic/memory contention. In order to consider this factor, the MFLUSH operational environment includes a *Multicore Traffic (MT)* delay, that is added to both MIN and MAX values as shown in Figure 6. The *MT* delay obeys the following equation:

$$MT = (L1_L2_Bus_delay + L2_Bank_Acc_delay)*$$

$$*(Num_Cores - 1)\ \textit{(cycles)}$$

Due to the high-variability of the L2 cache access time in CMP+SMT implementations sharing a single L2 cache, it cannot be used an static value to predict L2 cache misses, as done by the the FLUSH mechanism in SMT processors. For each L2 cache access, the MFLUSH mechanism predicts its resolution time according to the varying conditions of memory traffic and contention. The mechanism to obtain these predictions is described in Section 4.1. Based on each prediction, the MFLUSH dynamically estimates a *Barrier* value for that memory access. Whenever a memory access lasts more than *Barrier* cycles without being resolved it is considered to miss in the L2 cache. In that case, the FLUSH mechanism is triggered (See Figure 6), both stalling the offending thread and freeing some of its hardware resources (e.g., rename registers, instruction queue entries, etc). Exactly as in the FLUSH mechanism, the offending thread remains idle until the memory access is resolved. During this period of time, the freed resources, originally devoted to the newest instructions of the offending thread, may be used by all other running threads in the same SMT core. The *Barrier* estimation obeys the following equation:

$$BARRIER = \frac{L2prediction + MIN}{2} + MT\ \textit{(cycles)}$$

In presence of high memory traffic/contention, an *L2 cache hit* (*Late*) may be as harmful as an *L2 cache miss*. In that case, the *Barrier* value could be too high, involving a possible resource waste. In order to reduce the negative effects of *Late L2 hits*, the MFLUSH considers *suspicious* all L2 cache accesses that last more than $MIN + MT$ execution cycles to be resolved. As shown in Figure 6, the MFLUSH operational environment establishes a *Preventive State* for all *suspicious* memory accesses. Thus, any threads with a *suspicious* in-flight memory access is stalled by the MFLUSH mechanism, preventing it from obtaining additional hardware resources. However, a thread in the *Preventive State* is still running and can make forward progress with the instructions priorly fetched into the execution pipeline. Whether the *suspicious* memory access is resolved before reaching the *Barrier* the corresponding thread is removed from the *Preventive State*. In that case, the thread is allowed to fetch new instructions into the pipeline. Otherwise, the *suspicious* memory access is predicted as an L2 miss, and the FLUSH mechanism is triggered.

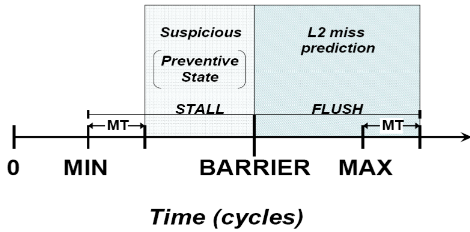


Figure 6. MFLUSH Operational Environment.

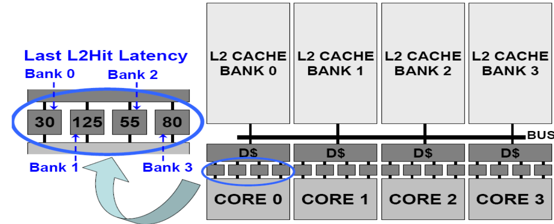


Figure 7. MFLUSH hardware support for a 4-core CMP with a 4-banked L2 Cache.

Triggering the FLUSH mechanism has a cost, both in terms of performance and power consumption. A flushed thread is stalled until the offending memory access (load instruction) is resolved, avoiding additional forward progress in the whole thread. Besides, all the newest instructions issued, from the last fetched instruction to the offending memory instruction, are flushed away from the execution pipeline. By the time the offending memory access is resolved, the thread resumes its execution, fetching again in the execution pipeline all flushed instructions. Consequently, all flushed instructions have a higher cost in terms of power consumption. The exact cost depends on the pipeline stage the instruction was by the time it was flushed. Therefore, making an smart use of the FLUSH mechanism is critical to obtain both good performance and a moderated power consumption.

4.1 MFLUSH Hardware Support

In order to obtain both fast and accurate dynamic predictions, the MFLUSH policy requires some additional hardware support, shown in Figure 7. Each SMT core holds an 8-bit register (*MCReg*) per each L2 cache bank used. The *MCReg* register keeps the latency of the last *L2 cache hit* in the corresponding L2 cache bank. The MFLUSH mechanism assumes the same behavior in consecutive accesses to the same L2 cache bank. Hence, the MFLUSH uses the value in the corresponding *MCReg* register to quickly predict the latency of the next access to the same L2 cache bank.

Figure 7 shows an example for a 4-core CMP implementation where all cores share a 4-banked L2 cache. Each core is connected to each of the L2 cache banks by means of a shared bus. In case of an L1 cache miss in core 0, the L2 cache bank that should contain the requested data is first determined using the address of the corresponding memory access. The MFLUSH mechanism then accesses the corresponding *MCReg* register and uses its content as prediction of the L2 hit latency. Thus, if bank 2 was acceded, the latency prediction would be of 55 cycles, as shown in Figure 7. Using this L2 hit latency prediction the MFLUSH mechanism proceeds with the appropriate response according to the varying memory traffic/contention conditions, as described in Section 4.

The *MCReg* registers admit more complex configurations, involving queues (i.e., history length : $MCReg = 1$; $queues > 1$) and more complex functions to determine the prediction from all queue entries. However, to keep it simple and fast we use a single *MCReg* register per core and per L2 cache bank. Our results confirm that this choice allows tracking quick memory behavior changes.

4.2 MFLUSH Throughput Evaluation

Figure 8 shows the system throughput evaluation for CMP+SMT implementations with 2, 3, and 4 cores, using 4, 6, and 8-thread workloads respectively. The results in Figure 8 include, for each workload, 4 evaluations using different IFetch Policies: *ICOUNT*, *Speculative FLUSH with 30-cycle trigger (FLUSH-S30)*, *Speculative FLUSH with 100-cycle trigger (FLUSH-S100)*, and *MFLUSH*. Figure 8 shows that in general, the highest results are obtained using *FLUSH-S100*. However, it is not true for all considered workloads, as in the case of *4W4*, *6W4*, and *8W1*, in which the *MFLUSH* yields the highest results. The results in Figure 8 also confirm that a bad trigger choice in *Speculative FLUSH*, as happens with *FLUSH-S30* (30 cycles) in most of the cases, may yield even worse results than the *ICOUNT* IFetch Policy. Examples of this situation are *4W1*, *6W1*, and *8W4*. Recall that this trigger choice yields an average 22% speed-up over *ICOUNT* in single-core SMT, as shown in Figure 2. Something similar occurs in the *4W3* workload, where the *ICOUNT* IFetch Policy yields 4% speed-up over *MFLUSH*. This isolated fact is due to the specific workload and microarchitecture characteristics.

Focusing on average results, it can be asserted from Figure 8 that the *MFLUSH* effectively succeeds in giving high throughput results, 2% close to the best performing *Speculative FLUSH* option (*FLUSH-S100*). This goal is achieved without requiring additional information regarding neither the trigger value to be used

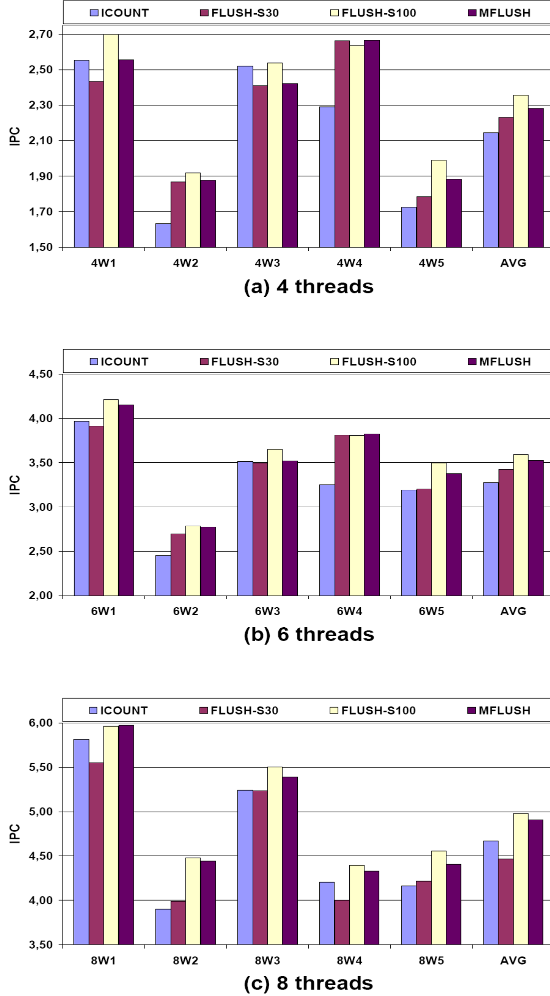


Figure 8. Throughput Results.

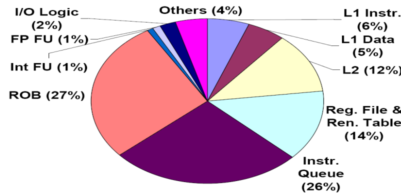
nor the underlying CMP+SMT implementation. Recall that *Speculative FLUSH* requires to specify apriori a trigger value (i.e., a 100-cycle trigger for the *FLUSH-S100*).

4.3 MFLUSH Power Consumption Evaluation

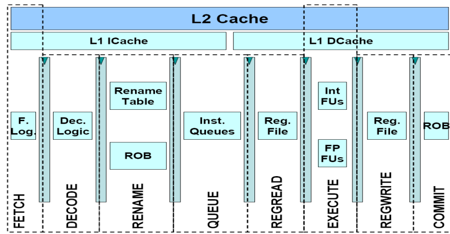
It is well-known that the *FLUSH* mechanism is a high-power-consumption alternative aimed at throughput-oriented scenarios, in which the system throughput is the main concern regardless of the power required. Flushing away instructions from the pipeline, and having to re-fetch them afterwards, implies an additional energy cost. This cost depends on the pipeline stage in which was the instruction by the flush time. According to the energy consumption analysis done in [5], Figure 9(a) shows the energy consumption distribution, per each hardware resource, in a typical execution pipeline. Assuming that each instruction requires 1 *energy unit* to be committed (the exact energy amount depends on the specific microarchitecture specifications), and given the resource usage through the execution pipeline stages shown in Figure 9(b) for a typical SMT core, Figure 10 shows the *Energy Consumption Factor*. This factor allows to estimate the additional energy required by the *FLUSH* mechanism, just tracking the number of flushed instructions in each pipeline stage and applying the corresponding factor value. Compared to FLUSH, the MFLUSH mechanism only adds a read access to a local 8-bit register on L1 cache misses. A write access to that register is only required in case of L2 hits. Due to its reduced cost, the MFLUSH hardware support is not added to the *Energy Consumption Factor*.

Nowadays, the power-aware constraints are present even for throughput-oriented scenarios. Although there are still scenarios in which obtaining the highest throughput is the main concern, the power constraints impose severe constraints on how this goal is achieved. Consequently, any architectural advance which reduces the energy consumption without hardly compromissing the total throughput is of particular interest.

Figure 11 shows the *Wasted Energy* implied by each *Speculative FLUSH* (*FLUSH-S30* and *FLUSH-S100*) and *MFLUSH* IFetch Policy. This *Wasted Energy* strictly corresponds to the additional energy required by the FLUSH mechanism, which requires re-fetching flushed instructions once resolved the corresponding memory accesses. The *Wasted Energy* is measured in *energy units* in Figure 11, that is the amount of energy required to commit 1 instruction. The results in Figure 11 are obtained using the *Energy Consumption Factor* (See Figure 10) and the number of instructions flushed in each pipeline stage. Figure 11 indicates that *FLUSH-S100* wastes in average 10% more energy than *FLUSH-S30*. Although *FLUSH-S100* involves less total flushes than *FLUSH-S30*, it involves more instructions to be reflushed. Waiting more time implies more instructions fetched into the execution pipeline by the time the FLUSH mechanism is triggered, and therefore a greater amount of instructions to refetch. Figure 11 also points out that aggressive flushing comes at an extra energy cost. In all cases the *MFLUSH* obtains significant energy consumption reductions, reaching 20% when compared with the best-performing *Speculative FLUSH* choice (*FLUSH-S100*), that obtains a marginal 2% throughput improvement over *MFLUSH*. Consequently, the *MFLUSH* IFetch Policy constitutes not only a solution to the unpredictability of the L2 cache latency in the emerging *CMP+SMT* scenario but also provides an important energy consumption saving.



(a) Distribution (% per resource).



(b) Pipeline Stages/Resources Distribution.

Figure 9. Energy Consumption.

	Energy Consumption Factor	
Pipeline stage	Local	Accumulated
Fetch	0,13	0,13
Decode	0,03	0,16
Rename	0,22	0,38
Queue	0,26	0,64
Reg. Read	0,05	0,69
Execute	0,13	0,82
Reg. Write	0,05	0,87
Commit	0,13	1

Figure 10. Energy Consumption Factor.

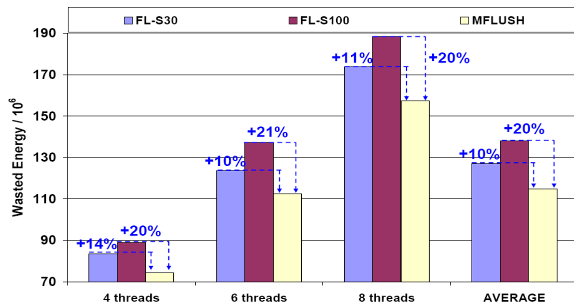


Figure 11. FLUSH Wasted Energy.

5 Related Work

The FLUSH mechanism was proposed by Tullsen et al. in [12] as an improvement for the ICOUNT [14] policy in single-core SMT processors. The ICOUNT policy has difficulties with threads that experience many loads that miss in L2, being unable to realize that a thread can be blocked on an L2 miss and do not make forward progress for many cycles. Depending on the amount of instructions dependent of the blocked load, many processor resources may be blocked and the total throughput suffers from a serious slow-down. Several FLUSH implementation choices were analyzed in [12], focusing on the simplest and less expensive ones : *Trigger on Delay* or *Speculative FLUSH*. With the rise of the emerging CMP comprised of SMT cores, like the IBM POWER5 [11] and POWER6 [6], it must be faced a new challenge: the unpredictability of the L2 cache hit latency. The MFLUSH mechanism adapts the FLUSH and STALL phylosophy to the new CMP+SMT scenario, obtaining both dynamic adaptability to the varying memory traffic/contention conditions and important energy consumption savings.

Several authors have shown that long latency operations have to be taken into account by the IFetch Policy in order to boost SMT performance [3, 4, 12, 14]. In order to apply them to the new *CMP+SMT* scenario a similar analysis, as done in this paper, should be performed. Revisiting prior well-known high-performance proposals when moving to a new application scenario generally requires this type of prior analyses.

Shin et al. propose an *Adaptative Dynamic Thread Scheduling* (ADTS) [10] to manage the resource sharing in SMT processors. The ADTS improves the system throughput in SMT processors by adapting the underlying IFetch Policy to the workload characteristics. Thus, the ADTS changes the IFetch Policy used among ICOUNT [14], BRCOUNT [14], and L1DMISSCOUNT [14], according to the varying workload characteristics. In this work we propose the MFLUSH mechanism that adapts the FLUSH and STALL philosophy to the emerging CMP+SMT scenario. The MFLUSH dynamically adapts to the varying memory traffic/contention conditions and additionally achieves important energy consumption savings.

6 Conclusions

In this paper we analyze the new challenges to be faced in future high-degree Multithreaded *CMPs*, with multiple *SMT* execution cores sharing an L2 cache (*CMP+SMT*). In particular we focus on probably the most important SMT issue: the *Instruction Fetch Policy*. Considering *ICOUNT* and *FLUSH* IFetch Policies we show results which evidence that *CMP+SMT* may not simply relly on *SMT* IFetch Policies to boost overall throughput. *SMT* IFetch Policies must be revisited when moving to the new *CMP+SMT* scenario.

From the exhaustive analysis included herein, it is proposed a novel IFetch Policy designed to cope with the emerging *CMP+SMT* scenario: the *MFLUSH*. We include a complete evaluation of the *MFLUSH*, both in terms of throughput and energy consumption. Our results indicate that the *MFLUSH* succeeds not only in overcoming the specific *CMP+SMT* constraints but

also allowing a 20% energy consumption reduction without a significative system throughput loss.

Acknowledgements

This work has been supported by the Ministry of Education of Spain under contract TIN2007-60625, the Barcelona Supercomputing Center(BSC) and the HiPEAC European Network of Excellence. The authors wish to thank Daniel Ortega, Ayose Falcon, Jeroen Vermoulen, and Oliverio J. Santana for their support and help with the simulation tools.

References

[1] UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007.

[2] UltraSPARC T1 Supplement. *Draft D2.0, 17 Mar*, 2006.

[3] F. J. Cazorla, E. Fernández, A. Ramirez, and M. Valero. Dynamically Controlled Resource Allocation in SMT Processors. In *Proc. of MICRO-37*, 2004.

[4] A. El-Moursy and D. H. Albonesi. Front-end policies for improved issue efficiency in SMT processors. In *Proc. of HPCA-9*, 2003.

[5] D. Folegnani and A. Gonzalez. Energy-Effective Issue Logic. In *Proc. of ISCA-28*, 2001.

[6] H. Le, W. Starke, J. Fields, F. O'Connell, D. Nguyen, B. Ronchetti, W. Sauer, E. Schwarz, and M. Vaden. IBM POWER6 microarchitecture. *IBM J. Res. Dev.*, 51(6):639–662, 2007.

[7] M. J. Serrano and R. Wood and M. Nemirovsky. A Study on Multistreamed Superscalar Processors. Technical Report 93-05, University of California Santa Barbara, 1993.

[8] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proc. of ASPLOS-7*, 1996.

[9] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *Proc. of PACT-10*, 2001.

[10] C. Shin, S-W. Lee, and J-L. Gaudiot. Dynamic scheduling issues in SMT architectures. 2003.

[11] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 System microarchitecture. *IBM J. Res. Dev.*, 49(4/5):505–521, 2005.

[12] D. M. Tullsen and J. A. Brown. Handling Long-latency loads in a Simultaneous Multithreaded Processor. In *Proc. of MICRO-34*, 2001.

[13] D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. of ISCA-22*, 1995.

[14] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. of ISCA-23*, 1996.

[15] Ofri Wechsler. Inside Intel Core Microarchitecture. *White Paper*.

[16] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Proc. of PACT*, 1995.