Introducing Runahead Threads for SMT Processors

Tanausú Ramírez[1], Alex Pajuelo[1], Oliverio J. Santana[2], Mateo Valero[1,3]

[1]Universitat Politècnica de Catalunya, Spain. {tramirez, mpajuelo}@ac.upc.edu.

[2]Universidad de Las Palmas de Gran Canaria, Spain. ojsantana@dis.ulpgc.es

[3]Barcelona Supercomputing Center, Spain. mateo.valero@bsc.es

Abstract

In this paper, we propose Runahead threads on Simultaneous Multithreading processors as a valuable solution for both exploiting the memory-level parallelism and reducing the resource contention. This approach transforms a memory-bounded eager resource thread into a speculative light thread, alleviating critical resource conflicts among multiple threads. Furthermore, it improves the thread-level parallelism by removing long-latency memory operations from the instruction window, and thus it releases busy resources otherwise. We compare an SMT architecture using Runahead threads (SMTRA) to both state-of-the-art static fetch and dynamic resource control policies. Our results show that the SMTRA combination performs better, in terms of throughput and fairness, than any of the other policies.

1 Introduction

Simultaneous Multithreading (SMT) [12][14] is a trendy architectural design based on executing multiple instructions from multiple threads at the same time. In this environment, threads not only share the resources, but they also compete for important resources in the processor core. Resource sharing in SMT is not easy and may significantly hinder the benefit of multithreaded execution. The different features and requirements of programs can cause that if a thread allocates too many resources, other threads could not have enough resources to continue executing instructions. The most harmful case occurs with threads presenting poor cache behaviors (memory-bounded threads). A memory-bounded thread can hold critical resources without making any progress due to long latency memory operations, since following instructions cannot be issued nor committed. This starves other threads of available resources and prevents them from advancing too. Consequently, this is a counterproductive effect that can lead to performance degradation in SMT processors.

Therefore, resource scheduling is an important trade-off because it determines how these resources should be shared. Nowadays, different fetch policies and resource schedulers have been proposed. A fetch policy decides which threads can feed the processor with new instructions to get the opportunity of using the available resources. A resource scheduler controls the resource allocation among threads trying to avoid the resource monopolization. These resource control policies stall or flush threads under determined conditions to prevent the resource overuse. Sometimes, these drastic actions, either stalling or flushing threads, can produce two negative effects. On the one hand, they may lead to a resource under-utilization situation, because they can prevent a stopped thread from using resources that no other thread requires. On the other hand, memory-bounded thread performance is virtually degraded to unfairly benefit fast ILP threads.

Our proposal in this paper is neither to let the thread excess in the use of resources, nor to keep stopping the thread. We propose Runahead threads to enable a thread for doing something beneficial to itself without disturbing the others threads. When a thread switches to a Runahead thread due to a long-latency load, it enters into an speculative light mode which requires the minimal amount of resources to execute. With this ability, Runahead threads allow memory-bounded threads speculatively going in advance without harming the other ILP threads. In this sense, Runahead threads improve the total performance by increas-

ing the memory level parallelism alleviating the resource contention among threads.

In this paper, we explain and discuss how the Runahead mechanism [5][7] can be adapted to SMT processors in order to make Runahead threads possible. Our evaluation shows that extending SMT with Runahead threads (SMTRA) improves the throughput for different kinds of workloads with regard to recent dynamic resource scheduling techniques. Likewise, this performance improvement is achieved with a significant overall balance between performance and fairness.

The paper is organized as follows: Section 2 discusses previous related work. Section 3 describes our experimental framework. We describe in detail in Section 4 how the Runahead mechanism is applied to SMT processors. We then evaluate the SMTRA proposal with regard to state-of-the art fetch and resource allocation policies in Section 5. Finally, we conclude in Section 6.

2 Related Work

In this section we comment the related work in two main research lines in SMT processors: resource scheduling mechanisms and speculative multithreading techniques. We discuss these areas because our proposal catch the essence of both: try to reduce the resources conflicts through an speculative mechanism. However, we simply reduce resource contention in a different way since, we do not make a detailed control of resources and do not use several contexts to improve the performance of only one thread.

2.1 Resource Scheduling

SMT processors in the literature include both fetch policies and resource allocation schedulers that work together to alleviate resource contention. Initial fetch policies, like *Round Robin* [12] and *ICOUNT* [12], only determine which threads feed the processor pipeline and which are left out, giving different priority to each thread. Next, since threads that experience many L2 cache misses can monopolize resources, several techniques built on top of ICOUNT were proposed. *STALL* [11] detects that a thread has a pending L2 miss and stalls the thread avoiding fetching further instructions. However, the L2 miss detection mechanism may be too late and a thread may hold many resources until the long latency operation is solved. *FLUSH* [11] minimizes this problem by flushing instructions of a thread with a long latency operation. The victim thread de-allocates all the resources, making them available to other executing threads.

The main drawback of previous policies is that they never control the per-thread resource utilization. They only take decisions based on static criteria or events to release resources. Therefore, at the time they try to prevent resource monopolization, they can also introduce resource under-use because take resources away that other threads do not need. Recently, some dynamic resource control policies have been proposed. *DCRA* [1] directly monitors the usage of resources by each thread trying to guarantee that all threads get their fair amount of the critical shared resources. *Hill Climbing* [3], instead of monitoring the resource indicators, varies the resource share of multiple threads toward the direction which improves the SMT performance (using the gradient descent algorithm with a function derived from performance metric). In both techniques, when a thread exceeds its assigned resource allocation, the thread is stalled until decrease its utilization.

2.2 Speculative Multithreading

Nowadays, novel thread-based prefetching techniques have gained a lot of interest in the research community due to new commercial Multithreading and Multicores architectures. In the context of SMT processors, several approaches (TME [13], SSMT [2] and DDMT [8]) try to exploit the thread level parallelism to speculate and execute various threads simultaneously. Most of these precomputation or pre-execution techniques directly execute a subset of the original program instructions on separate threads along the main computation thread. These additional threads (commonly called *helper-threads*) run ahead of the main thread and trigger cache misses earlier on its behalf, thereby hiding the memory latency.

Recent proposals [4] dynamically construct code slices (p-slices) via hardware, to execute in helper threads (p-threads) performing precomputation and prefetching. These techniques require construction of efficient p-slices. The main drawback of all the schemes mentioned above is that they employ several contexts and processor resources to enhance the performance of a single main thread. Although SMT framework sounds like an easy and direct solution, the overhead to

create, spawn the separate threads and communicating with the main thread can be complex and prejudicial for performance.

3 Experimental Framework

Our simulation environment is based on an SMT execution-driven simulator derived from SMTSIM [10]. We significantly modified the simulator to support simulation checkpoints, also including the Runahead mechanism and an enhanced memory hierarchy. In our SMT model, we use a complete resource sharing organization because it requires a minimal hardware complexity. Then, the threads coexist in the different processor stages, sharing the issue queues and the reorder buffer (ROB), the physical registers, the functional units and the caches. Table 1 lists the main configuration parameters of this simulated SMT processor setup.

Processor core	
Processor depth	10 stages
Processor width	8 way
Reorder buffer size	512 shared entries
INT/FP registers	320 / 320
INT/FP/LS queues	64 / 64 / 64
INT/FP/LdSt units	6 / 3 / 4
Branch predictor	Perceptron
Memory subsystem	
Icache	64 KB, 4-way, 1 cyc pipelined
Dcache	64 KB, 4-way, 3 cyc latency
L2 Cache	1 MB, 8-way, 20 cyc latency
Caches line size	64 bytes
Memory latency	400 cycles

Table 1: SMT processor baseline configuration

The experiments were performed with workloads created from the SPEC 2000 benchmark suite. To create the multithreaded workloads (see Table 2), we consider different groups with 2 and 4 threads, based on the L2 cache miss rate of each program simulated in a single-threaded context. All benchmarks were compiled on an Alpha AXP-21264 using the Compaq C/C++ compiler with the -O3 optimization level to obtain Alpha standard binaries. For the different benchmarks that compose the workloads, we select 300 millions representative instructions for each benchmark with reference inputs set using Sim-Point [9].

4 Runahead in SMT Processors

Runahead execution is a well-known speculative mechanism whose goal is bringing data and instructions into the caches. It was first proposed for in-order processors [5] to improve the data cache performance. It was later extended for out-of-order processor as a simple alternative to large instruction windows [7]. In our work, we extent the Runahead application field to SMT processors for both exploiting the memory-level parallelism and reducing the resource contention.

4.1 Runahead Operation

Runahead mechanism basically prevents the reorder buffer from stalling on long-latency memory operations by executing speculative instructions trying to follow the more likely program path. To do this, when a memory operation that misses in the L2 cache gets to the ROB head, it firstly takes a checkpoint of the architectural state. After taking the checkpoint, the processor assigns an invalid or bogus value to the destination register of the memory instruction that caused the L2 miss and enters in *runahead mode*. During runahead mode, the processor continues speculatively executing instructions and pseudo-retiring them out of the instruction window. All the instructions that operate over an invalid value will produce invalid results and they will be considered invalid too. The propagation of this invalid state is made using an invalid bit (INV) associated with each physical register. These invalid instructions are directly driven to commit stage to pseudo-retire once they are detected as invalid. The instructions that do not depend on the invalid value are executed as normal, except that they do not update the architectural state.

Once the blocking memory operation that started runahead mode is resolved, the processor rolls back to the initial checkpoint and resumes to the normal execution. As a consequence, all the speculative work done by the processor is discarded. Nevertheless, this previous execution is not completely useless. Runahead benefit comes from pre-execution of these speculative instructions that issue future memory operations improving prefetch efficiency for the data and instruction caches in the real execution.

4.2 Runahead Threads

Our objective to incorporate Runahead speculative execution in the SMT scope is to transform an eager resource thread in a light-consumer thread with fast instruction stream execution. When a thread is in runahead mode, it generally uses the different resources during short time. The invalid

ILP2	MIX2	MEM2	ILP4	MIX4	MEM4
apsi,eon	applu,vortex	applu,art	apsi,eon,fma3d,gcc	ammp,applu,apsi,eon	art,mcf,swim,twolf
apsi,gcc	art,gzip	art,mcf	apsi,eon,gzip,vortex	art,gap,twolf,crafty	art,mcf,vpr,swim
bzip2,vortex	bzip2,mcf	art,twolf	apsi,gap,wupwise,perl	art,mcf,fma3d,gcc	art,twolf,equake,mcf
fma3d,gcc	equake,bzip2	art,vpr	crafty,fma3d,apsi,vortex	gzip,twolf,bzip2,mcf	equake,parser,mcf,lucas
fma3d,mesa	galgel,equake	equake,swim	fma3d,gcc,gzip,vortex	lucas,crafty,equake,bzip2	equake,vpr,applu,twolf
gcc,mgrid	lucas,crafty	mcf,twolf	gzip,bzip2,eon,gcc	mcf,mesa,lucas,gzip	mcf,twolf,vpr,parser
gzip,bzip2	mcf,eon	parser,mcf	mesa,gzip,fma3d,bzip2	swim,fma3d,vpr,bzip2	parser,applu,swim,twolf
gzip,vortex	swim,mgrid	swim,mcf	wupwise,gcc,mgrid,galgel	swim,twolf,gzip,vortex	swim,applu,art,mcf
mgrid,galgel	twolf,apsi	swim,vpr			
wupwise,gcc	wupwise,twolf	twolf,swim			

Table 2: SMT simulation workload classification

instructions use very few resources since they are going to be pseudo-retired as fast as possible. In the case of other long-latency loads, they are also invalidated just like the load that started the runahead mode, remaining only the memory access as prefetch. The rest of valid instructions executed in runahead are usually short-latency instructions, which also use a minimum part of resources. Therefore, *Runahead threads* are much less aggressive than normal ones with the valuable SMT resources, taking and releasing them in short periods of time.

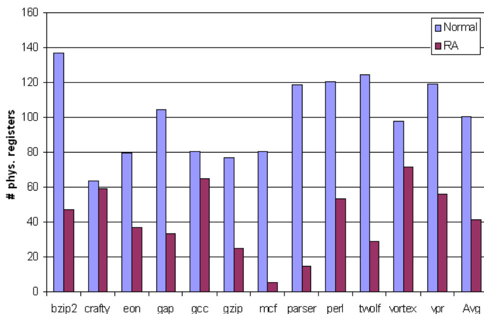


Figure 1: Average physical register used per cycle between the operation modes in RA for SPECINT 2000

To demonstrate this fact, Figure 1 shows the average amount of physical registers that each SPECInt2000 program keeps allocated per cycle. The figure has two bars per benchmark, the left bar shows physical registers allocated in normal mode and the right one shows physical registers allocated during runahead mode. These data show how the programs in runahead mode use much less than the half of registers used in normal execution. Memory-bounded benchmarks like *mcf* or *parser* use a very small quantity of physical registers while runahead execution is activated.

Therefore, in the context of SMT processor, our idea is that using this less eager resource mode for memory-bounded thread will allow them to going forward without limiting the available resources for other threads. At the same time, the issued prefetches in runahead make possible to increase the memory level parallelism of threads. Then, runahead execution lets the threads to do useful processing instead of stalling for several cycles due to resources contention as other techniques do.

4.3 SMT design issues for Runahead

At the moment of applying the Runahead mechanism in SMT processors, it is necessary to extend the implementation to several threads and bear some issues in mind. Now, we discuss some relevant design aspects related with Runahead mechanism in the SMT processors environment.

Register Invalidation. Each thread context usually has its own architectural registers per register file. We note that in our simulation model, we have fixed the total number of physical register instead of the number of rename registers. If we have a register file of 320 physical registers with an Alpha ISA, it means that the processor have a shared pool of $160 = 320 - (32x4)$ rename registers when 4 threads are running. To correctly recover the architectural state, each thread only needs to checkpoint the content of their architectural registers because a copy of the full physical register file is unnecessary and it would take a long time.

A Runahead thread also needs the INV bit vector to identify the validity of registers, since in case of an invalid one, its real value is not important. Then, to adapt the runahead operation to a multithreaded environment, each thread has its own INV bit vector to follow the propagation of their

registers invalidations. An instruction with an invalid operand is not executed and, when reaches the commit stage, it is pseudo-retired in program order. If it is a valid instruction, it updates its physical destination register and pseudo-retires after its execution. So, when a physical register is invalid (INV bit set to 1) this is candidate to be free and to be used soon for the rest of threads.

Thread Priority. There are now two sorts of threads that try to use the resources: normal threads (non-speculative) and Runahead threads (speculative). The question is whether the standard ICOUNT fetch policy is a correct scheme to manage this new situation. In order to analyze this new environment, we investigate several modification of the ICOUNT algorithm varying the thread priorities. However, our results show that ICOUNT is able of distributing effectively the threads in relation to the obtained throughput. Besides, it is the simplest option, not requiring additional logic or hardware complexity, and thus we select it as our fetch policy.

Floating-Point Operations. Generally, the computation of the address for memory accesses involves a base register plus an offset. This is an integer arithmetic operation, so floating-point (FP) instructions are not needed to compute these effective addresses. We base on this observation to decrease the resource demand of Runahead threads in our proposal for SMT. In this sense, we modify the mechanism to avoid the execution of FP instructions during runahead mode.

Although this modification was considered in [6] as an efficient optimization for runahead execution for out-of-order processor, we apply it here for an additional benefit in the SMT environment. If a Runahead thread does not execute FP instructions, it leaves free the floating-point resources of the SMT processor for the rest of non-speculative threads. With this modification, FP instructions do not occupy any processor resources in runahead mode after they are decoded.

Synchronization. Finally, one difference in the context of SMT processors is that there can be both independent and parallel programs. The latter normally uses a scheme that allows threads to synchronize within the processor. The basic mechanism relied on blocking, acquire, and release instructions to thread synchronization. In case that a parallel thread enters in runahead mode, these kinds of instructions should be ignored. Besides, the instructions inside the critical section are invalidated to avoid data inconsistency among parallel threads.

5 Comparative Evaluation

In this section, we evaluate the performance and fairness of our SMT proposal using Runahead threads (SMTRA) related with previous proposals for both instruction fetch policies and resource allocation control policies.

Here, we use two metrics for the Evaluation. One is the performance (IPC) throughput, measures as the average sum of IPC of all running threads in each workload:

$$Average_IPC = \frac{\sum_{i=1}^{n} IPC_{MT,i}}{n} \quad (1)$$

The other metric represents the fairness-performance balance which is the harmonic mean of IPC speedup of each thread compared to single-threaded execution:

$$Hmean = \frac{n}{\sum_{i=1}^{n} \frac{IPC_{ST,i}}{IPC_{MT,i}}} \quad (2)$$

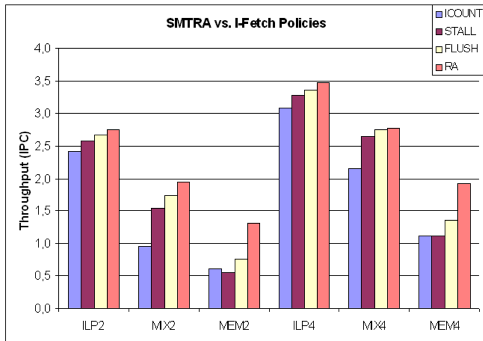
with $IPC_{MT,i}$ and $IPC_{ST,i}$ being the IPC fpr thread i in multithreaded and single-threaded mode respectively, and n being the number of threads.

5.1 SMTRA vs. I-fetch Policies

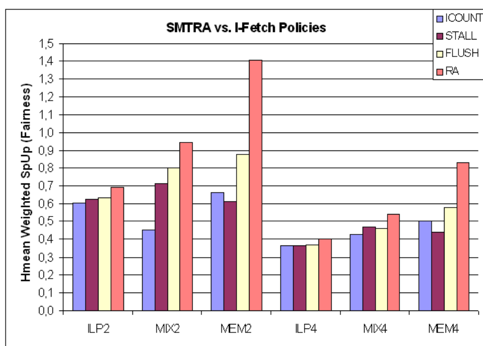
An instruction fetch (I-Fetch) policy determines from what thread to fetch instructions in a given cycle. Here, we evaluate two I-fetch schemes for handling long-latency loads, *STALL* and *FLUSH*, regarding SMTRA. All of them are implemented over the ICOUNT policy, which we use as the simple baseline reference.

Figures 2 shows the throughput (a) and the fairness (b) of these techniques for the different simulated workloads (see section 3). In general, from the performance point of view, flush outperforms stall, but SMTRA is clearly ahead of both.

In Figure 2(a), among the three types of workloads, SMTRA has the best performance mainly for memory-bound workloads: 69% and 40% better than flush for 2 and 4 threads respectively. The chance of SMTRA to exploit the memory-level parallelism in advance during runahead mode



(a) Average Throughput (IPC) of evaluated policies



(b) Hmean Weighted SpeedUp of evaluated policies

Figure 2: Throughput & Hmean relative to different workload for different I-Fetch policy.

causes this technique to fall into less stalls or flushs than the others, and then it does not slow down the program progress. Even, SMTRA gets throughhput improvement in ILP workloads. Although fetch policies can reduce the small conflicts in this fast workloads, avoiding future L2 misses have more benefits.

Figure 2(b) compares the fairness, defined in equation 2, of the different static techniques evaluated here. Again, SMTRA achieves the best results in terms of hmean in this case. Although, it does not obtain prominent improvements for ILP workloads (10% for ILP4), they are more impressive for MEM workloads: SMTRA gets 60% and 45% over flush for 2-thread and 4-thread workloads respectively. We also observe the fairness for both stall and flush is quite similar to ICOUNT for all 4-thread workloads. Even, stall loses 12% for MEM4.

5.2 SMTRA vs. Resource Control Policies

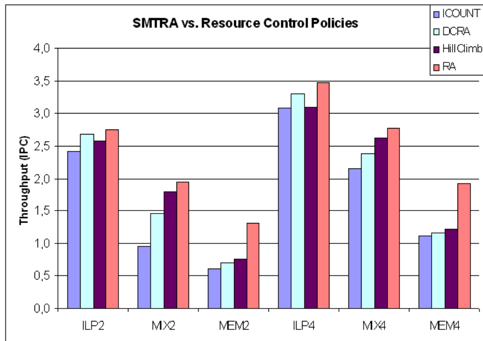
In the previous subsection, we evaluated some static long-latency aware fetch policies. Here, we compare SMTRA with two dynamic policies, which makes resource scheduling based on resource utilization (DCRA) or using an strategy guided by performance (HillClimbing) [1].

Figure 3 shows the IPC throughput (a) and harmonic mean (b) for ICOUNT (baseline), DCRA, HillClimbing and SMTRA respectively. As Figure 3(a) shows, all evaluated techniques perform better than the base ICOUNT. DCRA policy manages well the situation when there are ILP threads, and it slightly outperfoms HillClimbing (4% for ILP2 and 6.5% for ILP4) in these cases. For HillClimbing the fast execution changes of ILP workloads impedes the fine adjustment in their resource scheduling guided by the performance function. However, HillClimbing performs better than DCRA for MIX workloads (23% for MIX2 and 10% for MIX4), since it regulates well the different performance program characteristics to guide the resource requirements.

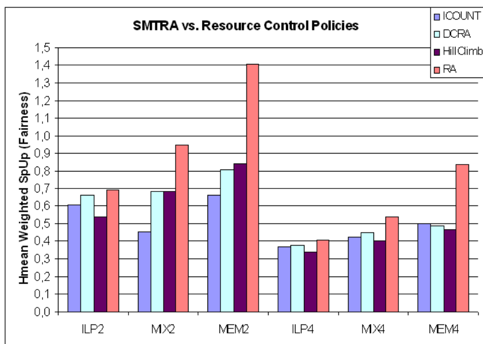
Nevertheless, we remark that SMTRA achieves higher throughput than any of the other resource control policies for all workloads. Like static policies, SMTRA performs excellent for MEM workloads. SMTRA improves DCRA throughput by 86% for MEM2 workloads and 65% for MEM4 workloads, and 69% and 58% respectively over HillClimbing. These results prove that it is preferable to try exploiting memory-level parallelism under overpressure on memory than strictly limiting the resources, even sometimes stalling the threads. The harm of a long-latency memory access has bigger impact than cycles penalties of resource conflicts. If we alleviate the former, we ease the latter, avoiding at the same time possible resource monopolization.

Regarding Hmean results, shown in Figure 3(b), SMTRA achieves better fairness than the others policies. Especially important is the fact that SMTRA considerably outperforms ICOUNT for all 4-thread workloads, while DCRA and HillClimbing lose fairness balance in some cases. Likewise, it is significant the average SMTRA hmean score for MEM workloads being 73% better than DCRA and 72% better than HillClimbing.

[1] Regarding HillClimbing, we use the performance function based on the throughput (named Hill-Thru in [3]



(a) Average Throughput (IPC) of evaluated policies



(b) Hmean Weighted SpeedUp of evaluated policies

Figure 3: Throughput & Hmean relative to different workload for different resource control policy.

Therefore, although SMTRA does not have knowledge of direct resource allocation among threads, it lets threads use a fair amount of resources to make speculative execution improving performance. The advantage comes from the right interaction between the fast runahead threads and normal threads, which make possible that both memory-bounded threads and the rest threads get improvements using the available resources.

However, we want to remark that the resource scheduling techniques evaluated in this subsection are orthogonal to the SMTRA combination proposed in this paper, *i.e.*, it is possible to incorporate an additional resource control mechanism to avoid possible inefficient resources utilization among normal and speculative threads. Logically, handling this new situation requires adaptation and modifications of the policies. We are studying and analyzing this approach as future work.

5.3 Extra Work

Unlike previous resource control techniques, the Runahead mechanism involves speculative execution that it is translated in a higher number of executed instructions. Flush is the only other technique that execute additional instructions, since it execute twice the issued instructions until the long-latency load detection point, in which these instructions are squashed. Therefore, the drawback of these techniques is that they generate extra instruction re-execution, increasing the overall energy consumption.

In this section we analyze the execution extra work ratio (EW) related with the useful work done in the SMT processor. We define this metric as the relation between the number instructions executed per each committed instruction. Then, this relation shows us the ratio of the additional speculatively instructions executed regarding the useful retired instructions to run a program. Likewise, it gives us an indirect power consumption level that a technique is producing.

In figure 4 we show the extra work (EW) for SMT with ICOUNT, Flush and Runahead mechanisms from left to right bars respectively. We note that, in this relation, it is also included the misspeculated instructions due to branch mispredictions, whose base portion can be observed in the ICOUNT bars. The higher a bar is, the worse.

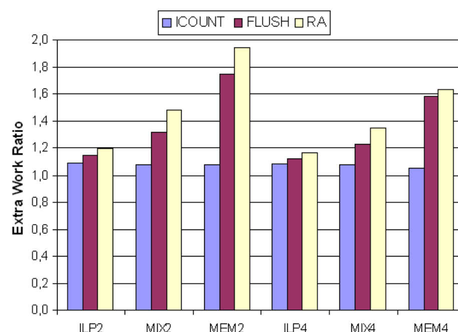


Figure 4: Execution Extra Work Ratio

We can see that SMTRA has a higher extra work ratio than Flush on average. SMTRA has 1,94 EW for 2-thread MEM workloads and 1,64 EW for 4-thread MEM workloads, while Flush has 1,75 EW and 1,59 respectively for the same workloads. Both techniques have much higher EW than

the rest of resource techniques evaluated in before sections, which they cannot produce speculative work except for mispredictions. Nowadays, the power consumption is an important limiting factor, therefore controls the useless extra work produced by Runahead threads is a topic which we were working on.

6 Conclusions

In this work we have presented Runahead threads, a speculative mechanism to ease the resource contention on Simultaneous Multithreading processors. This approach relies on applying runahead execution to different running threads when a long-latency load is pending. So, SMTRA allows extracting memory-level parallelism instead of stalling or flushing the threads as some previous work does in the SMT environment. In this way, this technique both avoids the possible resources monopolization from memory-bounded threads transforming them into light resource-demand threads and lets the others threads continue executing with the remainder resources.

This paper analyzes the different resource contention techniques, and compares them to the SMT with Runahead threads. We show the significant advantage of using SMTRA over these techniques both for performance improvement and fairness. Overall, SMTRA outperforms all of them, with improvements up to 86% on average. However, SMTRA is orthogonal to some dynamic resource scheduling schemes and it is possible to combine them to create more efficient schemes.

This work opens new ways to use Runahead threads or other speculative mechanisms as an alternative to resource contention in SMT processors. In addition, without forgetting the power consumption, there are still some interesting trade-offs on SMT processors to achieve better resource aware schemes.

7 Acknowledgments

This work has been supported by the Ministry of Education of Spain under contract TIN-2004-07739-C02-01 and grant AP2003-3682, the HiPEAC European Network of Excellence, and the Barcelona Supercomputing Center (BSC-CNS).

References

[1] F. J. Cazorla, A. Ramirez, E. Fernandez, and M. Valero. Dynamically controlled resource allocation in smt processors. In *MICRO-37*, pages 171–182, 2004.

[2] R. S. Chappell, J. Stark, S. K. Reinhardt, Y. N. Patt, and S. P. Kim. Simultaneous subordinate microthreading (ssmt). *ISCA-26*, 1999.

[3] S. Choi and D. Yeung. Learning-based smt processor resource distribution via hill-climbing. In *ISCA-33*, Washington, DC, USA, 2006.

[4] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. *MICRO'01*, 2001.

[5] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS-11*, New York, NY, USA, 1997.

[6] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *ISCA-32*, 2005.

[7] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA'03*, Washington, DC, USA, 2003.

[8] A. Roth and G. S. Sohi. Speculative data-driven multithreading. *HPCA'01*, 2001.

[9] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT'01*, Washington, DC, USA, 2001.

[10] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Int. Annual Computer Measurement Group Conference*, 1996.

[11] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *MICRO-34*, Washington, DC, USA, 2001.

[12] 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 *ISCA-23*, New York, NY, USA, 1996.

[13] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *ISCA-25*, Washington, DC, USA, 1998.

[14] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *PACT'95*, Manchester, UK, 1995.