Runahead Threads to Improve SMT Performance

Tanausú Ramírez[1], Alex Pajuelo[1], Oliverio J. Santana[2], Mateo Valero[1,3]
[1]Universitat Politècnica de Catalunya, Spain. {tramirez,mpajuelo,mateo}@ac.upc.edu.
[2]Universidad de Las Palmas de Gran Canaria, Spain. ojsantana@dis.ulpgc.es
[3]Barcelona Supercomputing Center, Spain.

Abstract

In this paper, we propose Runahead Threads (RaT) as a valuable solution for both reducing resource contention and exploiting memory-level parallelism in Simultaneous Multithreaded (SMT) processors. Our technique converts a resource intensive memory-bound thread to a speculative light thread under long-latency blocking memory operations. These speculative threads prefetch data and instructions with minimal resources, reducing critical resource conflicts between threads.

We compare an SMT architecture using RaT to both state-of-the-art static fetch policies and dynamic resource control policies. In terms of throughput and fairness, our results show that RaT performs better than any other policy. The proposed mechanism improves average throughput by 37% regarding previous static fetch policies and by 28% compared to previous dynamic resource scheduling mechanisms. RaT also improves fairness by 36% and 30% respectively. In addition, the proposed mechanism permits register file size reduction of up to 60% in a SMT processor without performance degradation.

1. Introduction

Simultaneous Multithreading (SMT) [18][21] is a modern architectural design based on executing multiple instructions from multiple threads at the same time. This paradigm focuses on sharing different processor resources to overlap execution of different threads to enhance performance. However, in SMT environments, threads not only share important resources, but also compete for them in processor core. The different characteristics and requirements of every thread can unbalance resource allocation. Some threads will consume more resources than others, degrading overall performance seriously and hindering the benefit of multithreaded execution.

To overcome this situation, different fetch policies and resource schedulers have been proposed. A fetch policy decides which threads can feed the processor with new instructions to exploit available resources. A resource scheduler controls the resource allocation among threads, trying to avoid resource monopolization. These resource control policies stall or flush threads under determined conditions to prevent resource overuse.

The worst case happens when threads with poor cache behavior (memory-bound threads) are executed. A memory-bound thread can block the reorder buffer, because following instructions can neither issue nor commit while waiting for long-latency memory operations. Besides, a lot of critical resources may have already been assigned to this thread, starving other threads of required resources and preventing their forward progress. This effect would likely lead to global performance degradation in SMT processors.

However, if we stall threads during memory intensive periods, they will advance too slowly. Consequently, the available memory-level parallelism is not exploited, failing to reach the potential performance achievable. Current resource policies do control actions, either stalling or flushing threads, which harm performance opportunities of memory-bound threads to unfairly benefit fast threads. Also, these policies can sometimes produce a resource under-utilization situation, preventing a stalled thread from using resources that no other thread requires.

In this paper, we propose *Runahead Threads* (RaT) to exploit this memory-level parallelism while reducing resource contention in SMT processors. Runahead[5][11] execution is a mechanism whose goal is to bring speculative data and instructions into the caches. We propose a new utilization of the Runahead mechanism on SMT processors as a different, fair memory-aware fetch policy to improve performance of memory-bound threads without harming ILP threads.

Our technique applies Runahead execution to any running thread when a long-latency load is pending. Thus, when a thread undergoes a long-latency load, it turns into a *runahead thread*. That is, it enters into a speculative light mode. While being a runahead thread, this thread uses the different resources

978-1-4244-2070-4/08/$25.00 ©2008 IEEE

during short time without limiting the available resources for other threads. At the same time, the issued prefetches increase the memory-level parallelism, improving its own performance. In other words, our proposal transforms an eager resource thread into a light-consumer thread with fast instruction stream execution. With this ability, *RaT* allows memory-bound threads to advance speculatively, instead of stalling the thread, doing beneficial work without disturbing the other threads.

Our evaluation shows that RaT improves throughput for different kinds of workloads with respect to previous I-fetch polices and dynamic resource scheduling techniques. This improvement is especially high for memory-bound workloads, which performs, on average, 83% and 65% better respectively. In addition, this throughput improvement is achieved with a significant overall balance between performance and fairness (70% and 56%) in a power-efficient way (better ED2). Finally, we show that the presented mechanism allows to reduce the size of the register file (up to 60%) of an SMT processor without penalizing performance.

The rest of this paper is organized as follows. Firstly, Section 2 discusses previous related work. We describe Runahead Threads in detail in Section 3. Section 4 describes our experimental environment and simulation tool. We evaluate RaT with regard to state-of-the art fetch and resource allocation policies in Section 5. In Section 6 we analyze the sources of improvement and the benefits of RaT. Finally, concluding remarks are given in Section 7.

2. Related Work

In the context of SMT processors, several speculative multithreading approaches try to exploit thread level parallelism by speculatively spawning threads: TME [20], SSMT [2] and DDMT [12]. Most of these pre-computation or pre-execution techniques directly execute a subset of the original program on separate processor contexts to speed up the main computation thread. These additional threads are commonly called *helper or assisted threads*. Recent proposals [4][22] dynamically construct code slices (p-slices) via hardware to execute in helper threads (p-threads) which perform pre-computation and prefetching. These techniques require the construction of efficient p-slices and the insertion of special instructions in the code.

Runahead Threads share some aspects with these pre-execution techniques. However, they use a different approach. They try to accelerate single running threads by spawning helper threads to pre-execute a shortened version of the program. Then, they employ several contexts and processor resources to improve the performance of that single main thread. Unfortunately, spawning separate threads and communication between them with the main thread can be complex and detrimental for performance.

On the contrary, our mechanism does not spawn new threads nor insert specific instructions. Our approach uses the same thread context to take benefit of long-latency useless memory access periods. In these periods, we apply Runahead Threads (RaT) to both exploit the memory-level parallelism and alleviate resource contention among threads. That is, we switch the critical thread into a light speculative thread that uses as few resources as possible to improve its performance.

SMT processors in literature include fetch policies and resource allocation schedulers that work together to alleviate resource contention. Initial fetch policies, like *Round Robin* [18] and *ICOUNT* [18], only determine which threads feed the processor pipeline and which are left out, assigning different priority to each thread. Several techniques built on top of ICOUNT were proposed later to prevent threads blocked by memory operations from monopolize the shared resources. *STALL* [17] detects that a thread has a pending L2 miss and stops fetching further instructions. The allocated resources are held until the L2 miss is solved. *FLUSH* [17] minimizes this problem by flushing the instructions of a thread during a long-latency memory operation. The victim thread de-allocates all of its resources, making them available to other executing threads at the cost of increasing its re-start latency.

These static policies never control the per-thread resource utilization. They only make decisions based on static criteria or events to release resources. Therefore, even if they try to prevent resource monopolization, they can sometimes cause resource under-utilization if other threads do not require the deallocated resource. 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 a fair amount of the critical shared resources. *Hill Climbing* [3], instead of monitoring the resource indicators, varies the resource allocation of multiple threads by using the gradient descent algorithm to improve throughput. In both techniques, when a thread exceeds its assigned resource sharing, it is stalled until that utilization decreases.

The MLP-aware fetch policy presented in [15] is a recent proposal that is related to our work. However, unlike our technique, it executes only some extra instructions indicated by a memory-level parallelism (MLP) predictor. After that, it stalls or flushes the thread. The number of speculative instructions executed is limited by the hardware setup of the MLP predictor (e.g. the long-latency shift register size). This limitation reduces the opportunities to improve performance, since not all distant MLP can be exploited.

Finally, a recent work [13] in the SMT context proposes a mechanism to release physical registers early belonging to those instructions that are independent of an L2 cache miss. The basis of this mechanism consists of traversing the ROB several times to identify which registers can be early deallocated (those that are not dependent on the L2-miss load). However, even if the idea is simple, the hardware overhead of the proposed mechanism is not when compared to our proposal.

3. Introducing Runahead Threads

Runahead execution is a well-known mechanism whose goal is to bring speculative 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 [11]. In this sense, Runahead consists of avoiding the blockage of the instruction window due to long-latency operations (*eg.* a load that misses in the L2-cache). Instead, the processor continues executing instructions speculatively, trying to follow the most likely program path until the load that triggered the runahead mode is resolved. The runahead benefit comes from the pre-execution of these speculative instructions which improves the data and instruction cache efficiency.

3.1. Runahead Operation

Runahead execution prevents the reorder buffer (ROB) from stalling on long-latency memory operations by executing speculative instructions. When a memory operation that missed in the L2 cache reaches the head of the ROB, a checkpoint of the architectural state is taken. After that, 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 the invalid value will be considered invalid. The propagation of this invalid state is made using an invalid bit (INV) associated with each physical register. These invalid instructions are folded (not executed) once they are detected as invalid. The instructions that do not depend on an invalid value are executed as normal, except that they do not update the architectural registers and memory state.

Once the memory access that started runahead mode is resolved, the processor rolls back to the initial checkpoint and resumes normal execution. As a consequence, all the speculative work done by the processor is discarded. Nevertheless, this execution is not completely useless since Runahead execution will generate useful data and instruction prefetches, improving the behavior of the memory hierarchy during real execution.

3.2. Runahead Threads

Resource conflicts are an important drawback for SMT processor performance. Our idea is to use Runahead Threads to transform a resource intensive thread (memory-bound thread) into a light-consumer thread with fast instruction stream execution. This allows threads to go forward, doing speculative prefetches without limiting the available resources for other threads.

When a thread is turned into a Runahead Thread, the invalid instructions do not use processor resources since they are pseudo-retired immediately. Other long-latency loads are also invalidated just like the load that started the runahead mode, performing the memory access only as a prefetch. The rest of the valid instructions executed in the runahead thread are usually short-latency instructions that use different resources for short periods of time. Therefore, Runahead Threads are much less aggressive than normal threads with the valuable processor resources, allocating and deallocating them in short periods of time. Besides, the issued prefetches in runahead increase the memory-level parallelism of threads. So, Runahead Threads allow the threads to do useful processing instead of stalling for several cycles due to resource contention.

3.3. Implementation details

The adaptation of runahead to SMT scenarios is not straightforward. Now, we describe some relevant design aspects related to Runahead Threads.

- Checkpoints. Each thread context usually has its own architectural registers per register file. To correctly recover the architectural state, each thread only needs to checkpoint the contents of its architectural registers. A copy of the full physical register file is unnecessary. Otherwise, the checkpoint would include the register information of all threads, taking long time.

- Register control. A runahead thread also needs the INV bit vector to identify the validity of computed values because, in case of an invalid one, its value is unimportant. Then, to adapt the runahead operation to a multithreaded environment, each thread has its own INV bit vector to track the propagation of register invalidations. An instruction with an invalid operand is not executed and when it 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. So, when a physical register is invalid (INV bit set to 1) this can be freed and used for the rest of the threads.

-Runahead cache. Mutlu *et al.* [11] introduce the runahead cache to provide communication of data and invalid status between runahead loads and stores. Based on this information, some loads dependent on stores can be identified as valid or invalid. Nevertheless, there are some cases in which this memory dependency cannot be identified. For example, a store that has an invalid effective address cannot save its status or data in the runahead cache.

From the SMT point of view, using a runahead cache can be expensive in terms of hardware. The runahead cache needs to be large to avoid line contention among threads. Likewise, it is necessary to include a new identification tag for each thread to distinguish the block owner. We measure the performance with and without the runahead cache to consider the need to include it in the RaT proposal and we found that using the runahead cache does not have significant impact on performance in our SMT model[1]. Based on this result and the fact that a runahead cache implies the use of more area in the SMT core, we decide not to use it in our RaT implementation. The functional difference is that some loads dependent on previous retired stores use stale values for speculative memory accesses, but it just affects Runahead execution, i.e., it does not affect correct program execution.

-Floating-point resources. Runahead Threads improve the performance of the SMT processor mainly due to the pre-execution of memory 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 the effective address. According to this observation, we can decrease the resource demand of Runahead Threads by avoiding the execution of FP instructions in Runahead Threads.

This modification was considered for runahead execution in out-of-order processors [10]. We apply it here again for an additional benefit in the SMT environment. If a runahead thread does not execute FP instructions, it does not need the floating-point resources of the SMT processor. So, once an instruction is detected to be an FP operation in the decode stage, it is invalidated and directly proceeds to pseudo-commit. With this modification, FP instructions in a runahead thread do not use any processor resources after they are decoded. Therefore, the FP issue queue, the FP functional units, and the FP physical register file are not used by most FP runahead instructions. The exceptions are FP loads and stores, which are treated as prefetch instructions because their effective addresses are obtained in the integer pipeline.

[1]In [11], the performance deviation without RA cache in SPEC2000 is also very small for a single-threaded out-of-order processor.

- Synchronization. Finally, an important issue 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 each other within the processor. The basic mechanism relies on block, acquire, and release instructions to perform thread synchronization. In the case that a parallel thread switches to a runahead thread, these instructions are ignored. The instructions inside the critical section are speculatively executed but do not modify program state avoiding data inconsistency among parallel threads.

4. Experimental Framework

Our simulation environment is based on an SMT execution-driven simulator derived from SMTSIM [16]. We have extended the simulator to support simulation checkpoints and a more precise memory hierarchy. We have implemented Runahead Threads in this simulator, as well as other fetch and resource scheduling techniques for comparison purposes.

Current SMT models use dynamic resource partitioning, which allows threads to improve their performance by allocating idle shared resources. This cannot be done in the case of a statically partitioned or monolithic SMT design, since each context has a fixed resource pool assigned. Thus, statically partitioned designs lead to a lack of flexibility far from the SMT ideal. In our SMT model, we use a complete resource sharing organization to benefit from the dynamic design advantages. The threads coexist in the different processor stages, sharing the issue queues, 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.

Table 1. SMT processor baseline configuration

Processor core	
Processor depth	10 stages
Processor width	8 way
Reorder buffer size	512 shared entries
INT/FP registers	320 / 320
INT/FP/LS issue 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
Main memory latency	400 cycles

As we can observe, the simulated processor uses a shared ROB for all the hardware threads. Using a separate ROB per thread would probably require less hardware complexity to implement Runahead Threads.

Table 2. SMT simulation workload classification

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			

However, we have chosen the shared ROB design to expose the mechanism to any possible critical resource contention. The additional complexity mainly lies in selectively squashing the speculative instructions once the runahead mode finishes. Nevertheless, this procedure is already implemented in an SMT processor with a shared ROB to recover from branch mispredictions.

The experiments were performed with workloads created from the SPEC 2000 benchmark suite. 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 each benchmark, we select an interval of 300 million instructions representative of entire program execution using the reference input set. To identify the most representative simulation point, we have analyzed the distribution of basic block execution using SimPoint[14]. Measurements are then taken using the FAME[19] evaluation methodology. FAME re-executes all traces in a multithreaded workload until all of them are fairly represented in the final measurements.

To create the multithreaded workloads, we consider only workloads composed of 2 or 4 threads. Several studies [6][8] have shown that SMT performance saturates or even degrades for workloads with more than 4 threads. We characterize the benchmarks based on the L2 cache miss rate of each program simulated in a single-threaded processor. Next, we group them into three types of workloads: high instruction-level parallelism threads (ILP), memory-bound threads (MEM), and a mixture of both (MIX). Table 2 shows our simulation workloads identified by the number of threads they contain and the thread types. Note that we choose large groups of workloads to avoid result deviation due to the specific behavior of a particular workload.

5. Comparative Evaluation

Now, we evaluate the performance and fairness of RaT compared to previous proposals based on instruction fetch and resource control policies.

We use two metrics for the evaluation. One is the performance (IPC) throughput, measured as the average sum of IPC of all running threads in a workload:

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

The other metric represents the fairness-performance balance, proposed in [9], which is the harmonic mean of IPC speedup of each thread compared to its single thread performance:

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

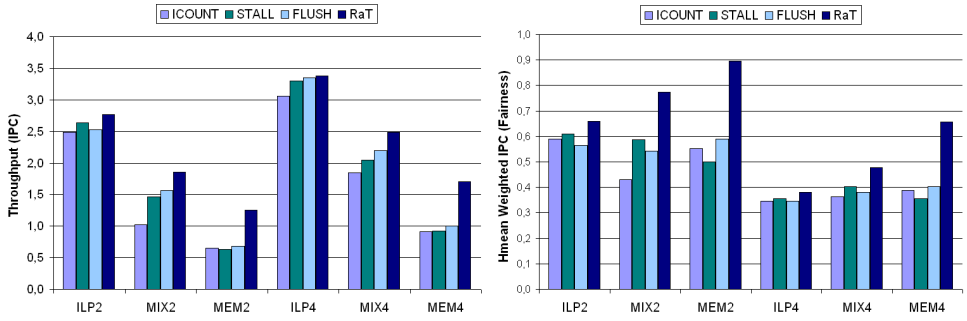
with $IPC_{MT,i}$ and $IPC_{ST,i}$ being the IPC for thread i in multithreaded and single-threaded mode respectively, and n being the number of threads.

5.1. RaT and I-Fetch Policies

An instruction fetch (I-Fetch) policy determines which thread is going to fetch instructions in a given cycle. We compare two I-fetch schemes for handling long-latency loads, *STALL* and *FLUSH* [17], with RaT. All of them are compared to the ICOUNT policy, which we use as the reference baseline.

Figure 1 shows the throughput (a) and the fairness (b) of these techniques for the different workloads. From the performance point of view, FLUSH outperforms STALL, but RaT is clearly ahead of both. In Figure 1(a), among the three types of workloads, RaT has the best performance mainly for memory-bound workloads, namely 83% and 70% better than FLUSH for 2 and 4 threads respectively. The fact that RaT exploits the memory-level parallelism during runahead mode avoids the need for stalling or flushing a thread, and thus it does not slow down the program progress.

Figure 1(b) compares the fairness of the different static techniques evaluated here. In this Figure, a higher bar is interpreted as better. Again, RaT achieves the best results in terms of fairness. Although the performance improvement for ILP workloads is moderate (10% for ILP4), it is more impressive for MEM workloads: RaT gets 55% and 63% improvement over FLUSH for 2-thread and 4-thread workloads respectively. We also observe that the fairness for both STALL and FLUSH has little deviation, being close to ICOUNT for all 4-thread workloads. Furthermore, STALL loses around 10% for MEM workloads.



(a) Throughput (IPC) (b) Fairness

Figure 1. Throughput and Fairness relative to workloads for different I-Fetch policies.

5.2. RaT and Resource Control Policies

Here, we compare RaT with two dynamic policies. DCRA [1] that makes resource scheduling decisions based on resource utilization and HillClimbing [3] that follows a strategy guided by performance. Regarding HillClimbing, we use the performance function based on the throughput (named Hill-Thru in [3]). The other two possibilities for the performance function (with weighted speedup and harmonic mean) use the IPC of each benchmark as a single thread as an external input. In this sense, we consider these options for HillClimbing highly dependent on the variability of the program characteristics. They require the repetition of the single program execution for different inputs and each thread in a real multithreaded processor to guide the mechanism.

Figure 2 shows the IPC throughput (a) and fairness (b) for ICOUNT (baseline), DCRA, HillClimbing and RaT respectively. As Figure 2(a) shows, all evaluated techniques perform better than the baseline ICOUNT. The DCRA policy manages the situation well when there are ILP threads, and slightly outperforms HillClimbing (4% for ILP2 and 5% for ILP4) in these cases. For HillClimbing the fast execution of phases of the ILP workloads avoids a fine adjustment in the performance function, which controls the resource scheduling. However, HillClimbing performs better than DCRA for MIX workloads (14% for MIX2 and 12% for MIX4), since it better controls the different performance characteristics of the programs to guide the resource requirements.

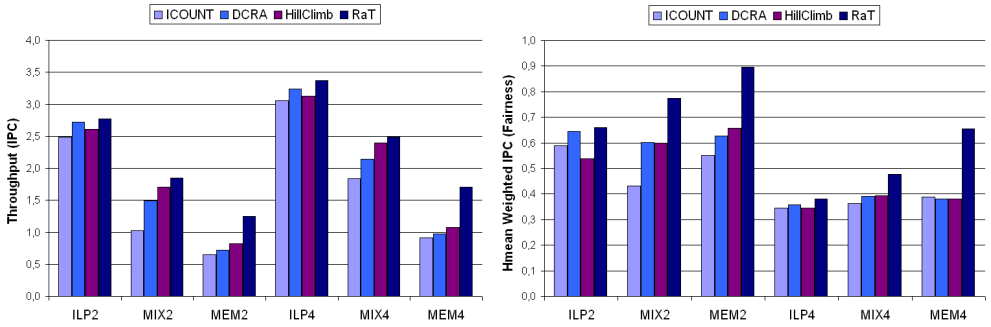
Nevertheless, we remark that RaT achieves higher throughput than any of the other resource control policies for all workloads. Like static policies, RaT increases considerably the performance of MEM workloads. RaT has a throughput improvement of 75% for MEM2 workloads and 74% for MEM4 workloads over DCRA, and 53% and 58% respectively over HillClimbing. These results (65% on average) prove that it is more preferable to exploit the memory-level parallelism than strictly limiting the resources or stalling the threads. The penalty of a long-latency memory access has a bigger impact than the cycle penalties of resource conflicts. If we alleviate the former, we ease the latter, while avoiding any possible resource monopolization.

In the case of fairness, shown in Figure 2(b), RaT achieves better results than the other policies. Essential is the fact that RaT considerably outperforms ICOUNT for all 4-thread workloads, whereas DCRA and HillClimbing lose fairness in some cases. Likewise, RaT fairness is also quite good for MEM workloads: 57% better than DCRA and 54% better than HillClimbing.

Therefore, although RaT does not have any knowledge about the direct resource allocation among threads, it lets threads use a fair amount of resources to allow speculative execution to improve the performance. The advantage comes from the right interaction between the fast runahead threads and normal threads, which means that both memory-bound threads and the other threads obtain performance improvements using the available resources. However, we want to remark that the techniques evaluated in this subsection (both DCRA and HillClimbing) are orthogonal to the mechanism proposed in this paper, *i.e.*, it is possible to incorporate an additional resource control mechanism to avoid possible inefficient resource utilization among normal and speculative threads. Logically, handling this new situation requires the adaptation and modification of the policies. We leave this study for future work.

5.3. Efficiency: Performance and Energy

Runahead Threads involve speculative execution, which translates into a higher number of executed instructions. Among the previously evaluated tech-



(a) Throughput (IPC) (b) Fairness

Figure 2. Throughput and Fairness relative to workloads for different resource control policies.

niques, FLUSH is the only other technique that executes additional instructions. This is due to the fact that FLUSH executes twice the instructions issued until the long-latency load detection point is reached, at which point these instructions are squashed. Therefore, the drawback of these techniques is that they generate extra instruction re-execution, increasing the overall energy consumption.

We evaluate the efficiency of the performance gain achieved and the additional energy consumed by each technique. To measure this efficiency, we use the commonly accepted Energy-Delay^2 metric [7]. This metric relates the processor power consumption to its performance. In our case, we measure the energy as the number of executed instructions. Although the wasted energy depends on the particular instructions executed, we assume that all the instructions consume the same amount of energy to simplify the analysis. The delay is counted as the average CPI, leading to this formula:

$$ED^2 = Num\_Executed\_Instr. * CPI^2$$

This formula provides an approximation of how efficiently the instructions are executed in terms of energy consumption. In Figure 3 we show the ED^2 for the evaluated techniques from left to right for each group of workloads. The bars are normalized to the ICOUNT values. Each bar may be interpreted as the higher the bar is, the more energy that is wasted per executed instruction compared to ICOUNT.

As shown in Figure 3, in spite of executing extra instructions, RaT provides excellent ED^2 results for the evaluated mechanisms. On average, RaT has 0.6 ED^2 for 2-thread workloads and 0.78 ED^2 for 4-thread workloads with regard to ICOUNT, while FLUSH presents 0.78 ED^2 for both workloads. In the MIX4 workloads, HillClimbing achieves better ED2 due to its good throughput and the smaller number of executed instructions. Except for this particular case,

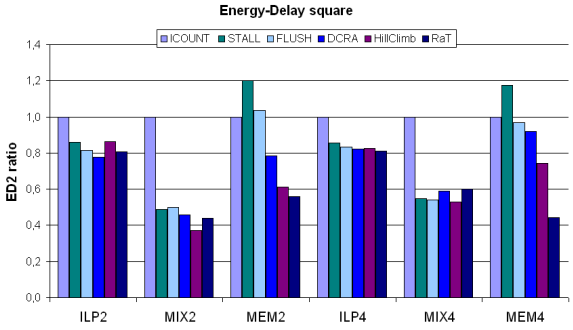


Figure 3. Energy-Delay2

the SMT processor consumes less energy per committed instruction using RaT while obtaining better performance.

6. RaT Benefits

Once RaT has been evaluated, we evaluate the sources of its benefit. One interesting point is how much threads are being improved by using Runahead Threads. This overall improvement comes from two distinct factors: (i) each thread itself is faster because of the *prefetching effect* via Runahead execution and (ii) Runahead Threads *release resources* to other threads. The former increases the memory-level parallelism whereas the latter reduces resource contention. In this section we make an analysis to distinguish the partial contribution of these two important factors, and complete the study with the register file impact.

6.1. Sources of Improvement

To isolate the sources of improvement of the proposed mechanism we have performed the following experiments:

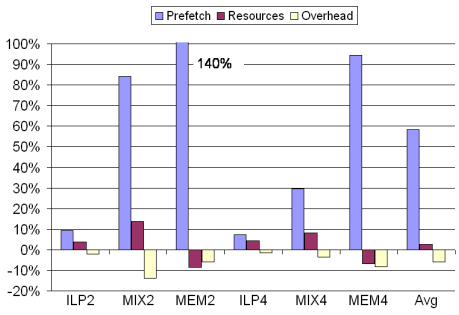


Figure 4. Sources of Improvement of RaT

-Prefetching. First, to measure the benefit of prefetching, we disable any access to the L2 cache during Runahead. Threads effectively turn into RaT but they perform no prefetch. In addition, loads and branches are tracked during Runahead mode to ensure that the runahead periods are the same in both normal RaT and RaT without prefetching. So, L2 miss loads found during RaT (without prefetching) will not switch again to RaT when they are encountered after recovering from Runahead.

The leftmost bar in Figure 4 shows the performance improvement of RaT compared to RaT without prefetching. Prefetch accounts, on average, for about 58% of the performance improvement. MIX and MEM workloads are the ones that benefit the most from this effect (56% and 109% respectively).

-Resource Availability. The resource availability comes from two points. Invalid instructions during Runahead do not hold resources because they are not executed. Secondly, instructions executed in Runahead present short latencies, meaning that registers are allocated for short periods of time.

In the next experiment, we prevent threads in Runahead mode from fetching more instructions when a load missing in the L2 cache is detected. Therefore, after the execution of already fetched instructions, RaT will not execute any more instruction, allowing other threads to exploit the available resources. The result of this experiment will serve as a measure of the benefit of early resource release by RaT.

The second bar in Figure 4 shows the performance improvement of this source of benefit. Although on average, the resource availability seems to improve the overall performance marginally (3%), this positive effect is mainly important for MIX workloads (22%). From the ILP-threads point of view, RaT behave like the flush mechanism for memory-bound threads: they release resources associated to invalid instructions while the L2 miss is being served. This fact enhances the execution of ILP threads since more resources are available.

-Overhead. Finally, we present the possible raw overhead of Runahead execution in SMT processors. In this case, we try to determine if the extra work performed in Runahead mode could disturb the other threads in the processor. As an approximated measure of this overhead, we examine if the remaining threads in the processor have any performance degradation when a RaT is executing without prefetched data. Notice that this is the worst case scenario since all the speculative work during Runahead mode is useless (no prefetch is performed and a lot of extra speculative work is executed).

The rightmost bar in Figure 4 shows that there is only a negligible 4% performance degradation. This demostrates that the periods of Runahead do not hurt the performance of the processor even if no useful prefetching is performed. As a result, RaT gets the full benefit of prefetching, without observing any effects of interference from the Runahead threads.

6.2. Impact on Register File

To complement the previous experiments, we select the register file as a case study of early resource release advantages obtained by RaT. Figure 5 shows the average amount of allocated physical registers per cycle for each kind of workload. There are two bars per workload. The left bar shows the average number of allocated physical registers per cycle in normal mode. The right bar shows the average number of allocated physical registers per cycle in runahead mode. This data clearly show that programs in runahead mode use less registers than in normal execution. In particular, memory-bound workloads with RaT use less than half the number of registers they would use without this mechanism.

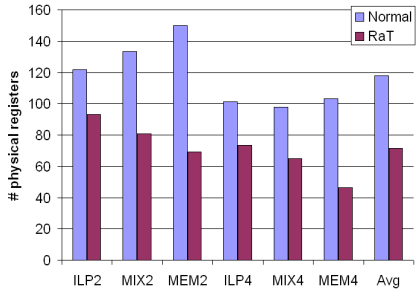
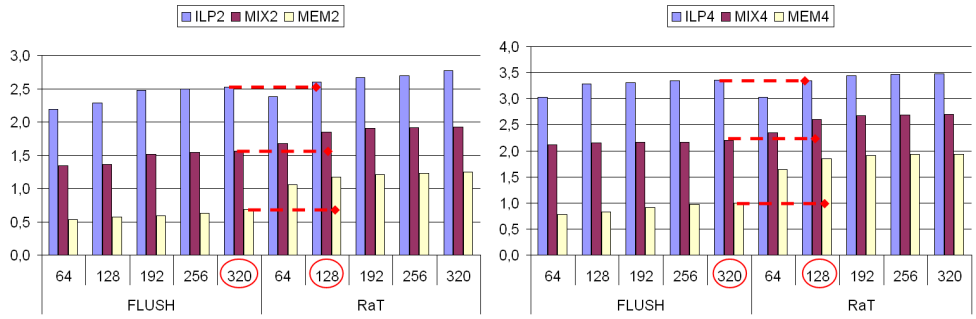


Figure 5. Average physical registers used per cycle between normal and RA modes

The register file is an important shared resource inside an SMT processor, and its size is one of the key issues in SMT design. This parameter, related to the ISA, sets the number of threads that an SMT processor is able to simultaneously execute in its core. With N



(a) 2-thread Workloads

(b) 4-thread Workloads

Figure 6. Throughput (IPC) relative to workloads for different register file size.

logical registers, N physical registers are needed and reserved for keeping the precise state of every thread. For example, in a 4-threaded SMT with an Alpha ISA (like our model) 32*4 = 128 physical registers are needed to maintain the architectural state of each thread. The rest of physical registers (i.e. 320 − 128) are available for sharing between threads. Then, the number of renaming registers must be large enough to support several simultaneous threads. However, larger register files have higher access times and a more complex design.

In this sense, we show that RaT additionally allows reducing the size of the register file. Following this, we compare the throughput of an SMT processor with the FLUSH policy (which also releases registers) to an SMT with RaT reducing the register file size. Figures 6(a) and 6(b) show the throughput for the 2-thread and 4-thread workloads respectively as a function of the register file size (from 64 to 320 registers). As we can observe, the throughput decreases when the number of registers is reduced, especially in the case of 4-thread workloads. However, this reduction is less dramatic when the RaT mechanism is used. For instance, the MEM4 workloads with FLUSH suffer from 27% slowdown passing from 320 registers to 64. When runahead threads are applied the slowdown is only 15%. Therefore, an SMT processor with RaT is less sensitive to register file size.

On the other hand, if we compare the throughput of FLUSH for all configurations to RaT with 64 registers, the latter overcomes the former for almost all combinations except for the ILP workloads. In fact, the performance of RaT using 128 physical registers is better than the performance of FLUSH using 320 physical registers (reduction of 60%). The performance improvement is 4%, 20% and 85% for ILP, MIX and MEM 2-thread workloads while is 0.2%, 21% and 92% for ILP, MIX and MEM 4-thread workloads respectively. Therefore, RaT allows using smaller register files without degrading performance, since it reduces the time a register is allocated making it available sooner to other instructions.

7. Conclusions

Memory-bound threads can monopolize resources in SMT processors without making any progress due to long-latency memory operations, degrading the performance of the other threads in the processor. Current fetch policies and resource control schemes usually restrict memory-bound threads in order to get higher throughputs. However the performance of fast threads is improved at the cost of degrading the performance of slow memory-bound threads.

Both Runahead and SMT processors are well-known microarchitectural techniques. Nevertheless, they are two different and clearly separate techniques that have not been considered together before. In this paper we propose joining both mechanisms to improve the performance of memory-bound threads without prejudicing fast threads.

Runahead Threads are an alternative solution to alleviate the resource contention in SMT processors. RaT avoids the possible resource monopolization of memory-bound threads, transforming them into light resource-demand threads and allowing the other threads to continue executing with the remaining resources. At the same time, memory threads get an important improvement from prefetching.

We evaluate and compare RaT with state-of-the-art resource contention techniques. In all cases, we show the significant advantages of using RaT over these techniques, especially for memory-bound workloads: RaT has 83% better throughput over static fetch policies (FLUSH and STALL). In the case of the performance/fairness balance, RaT shows 70% improvement

compared to fetch policies for MEM workloads. Likewise, RaT shows 65% and 56% better throughput and fairness compared to recent dynamic resource scheduling schemes. Overall, RaT outperforms all of them, with improvements up to 83%.

RaT is also more power-efficient than previous resource-aware SMT mechanisms. The extra power consumption is well-balanced for better global performance, which shows good efficiency in terms of ED2 (0.7) compared to other techniques. Moreover, it is interesting to note that RaT makes it possible to use smaller register files without degrading performance via a simple checkpoint implementation. Therefore, we conclude that RaT is an interesting choice for SMT processor designs that would influence the way in which future SMT processors balance resource usage between ILP and memory-bound threads.

Acknowledgments

This work has been supported by the Ministry of Education of Spain under grant AP2003-3682 and project contracts TIN-2004-07739 and TIN2007-60625, the HiPEAC European Network of Excellence, and the Barcelona Supercomputing Center (BSC-CNS).

We would like to thank Dean Tullsen for his useful suggestions and help on improving this work. We also like to thank Rick Strong, Ramon Canal, and Manoj Gupta for their help in preparing the final version of this manuscript.

References

[1] F. J. Cazorla, A. Ramirez, E. Fernandez, and M. Valero. Dynamically controlled resource allocation in smt processors. In *Int. Symposium on Microarchitecture (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). *Int. Symposium on Computer Architecture (ISCA-26)*, 00, 1999.

[3] S. Choi and D. Yeung. Learning-based smt processor resource distribution via hill-climbing. In *Int. Symposium on Computer Architecture (ISCA-33)*, pages 239–251, Washington, DC, USA, 2006.

[4] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. *Int. Symposium on Microarchitecture (MICRO-34)*, 2001.

[5] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Int. Conference on Supercomputing (ICS-11)*, NY, USA, 1997.

[6] R. Gonçalves, E. Ayguade, M. Valero, and P. Navaux. Performance evaluation of decoding and dispatching stages in simultaneous multithreaded architectures. *SBAC-PAD*, 2001.

[7] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE J. Solid-State Circuits*, pages Vol. 31, No. 9, 1996.

[8] S. Hily and A. Seznec. Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading. Technical Report PI-1086, INRIA, 1997.

[9] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in smt processors. In *ISPASS*, Tucson - AZ, USA, 2001.

[10] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *International Symposium on Computer Architecture (ISCA-32)*, pages 370–381, 2005.

[11] 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 *Int. Symposium on High-Performance Computer Architecture (HPCA-9)*, page 129, Washington, DC, USA, 2003.

[12] A. Roth and G. S. Sohi. Speculative data-driven multithreading. *Int. Symposium on High-Performance Computer Architecture (HPCA-7)*, 2001.

[13] J. Sharkey and D. Ponomarev. An l2-miss-driven early register deallocation for smt processors. In *Int. Conference on Supercomputing (ICS-21)*, pages 138–147, NY, USA, 2007.

[14] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Parallel Architectures and Compilation Techniques (PACT-10)*, pages 3–14, Barcelona, Spain, 2001.

[15] E. Stijn and E. Lieven. A memory-level parallelism aware fetch policy for smt processors. In *Int. Symposium on High-Performance Computer Architecture (HPCA-13)*, 2007.

[16] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Int. Annual Computer Measurement Group Conference*, pages 819–828, 1996.

[17] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Int. Symposium on Microarchitecture, 2007 (MICRO-34)*, Washington, DC, USA, 2001.

[18] 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 *Int. Symposium on Computer Architecture (ISCA-23)*, pages 191–202, New York, NY, USA, 1996.

[19] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. Fame: Fairly measuring multithreaded architectures. In *Parallel Architectures and Compilation Techniques (PACT-16)*, 2007.

[20] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *Int. Symposium on Computer Architecture (ISCA-25)*, pages 238–249, Washington, DC, USA, 1998.

[21] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Parallel Architectures and Compilation Techniques (PACT-4)*, pages 49–58, Manchester, UK, 1995.

[22] W. Zhang, D. M. Tullsen, and B. Calder. Accelerating and adapting precomputation threads for efficient prefetching. In *Int. Symposium on High-Performance Computer Architecture (HPCA-13)*, 2007.