Introducing Runahead Threads

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. mateo.valero@bsc.es

Abstract

Simultaneous Multithreading processors share their resources among multiple threads in order to improve performance. However, a resource control policy is needed to avoid resource conflicts and prevent some threads from monopolizing them. On the contrary, resource conflicts would cause other threads to suffer from resource starvation degrading the overall performance. This situation is especially sensitive for memory bounded threads, because they hold an important amount of resources while long latency accesses are being served. Several fetch policies and resource control techniques have been proposed to overcome these problems by limiting the per-thread resource utilization. Nevertheless, this limitation is harmful for memory bounded threads because it restricts the memory level parallelism available that hides the long latency memory accesses.

In this paper, we propose Runahead threads on SMT scenarios as a valuable solution for both exploiting the memory-level parallelism and reducing the resource contention. This approach switches a memory-bounded eager resource thread to a speculative light thread, avoiding critical resource blocking among multiple threads. Furthermore, it improves the thread-level parallelism by removing long-latency memory operations from the instruction window, releasing busy resources. 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) [17][19] is a trendy architectural design based on executing multiple instructions from multiple threads at the same time. The higher throughput of instructions obtained, the better overall SMT performance. This paradigm focuses on exploiting two kinds of parallelism: intra-thread instruction level parallelism (ILP) and inter-thread level parallelism (TLP). The former is obtained from the existing instruction level parallelism of each thread alone. The latter comes from overlapping the execution of different threads.

SMT processors achieve these targets by giving threads complete access to resources. Then, threads not only share the resources, but they also compete for them in this multithreaded environment. Instructions from different threads share and contest for important resources in the processor core like the fetch engine, the physical register file, the functional units, and the reorder buffer. Resource sharing in SMT is not easy and may significantly hinder the benefit of multithreaded execution. The different features and requirements of applications 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 allocate 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 to improve the performance. These resource control policies stall or flush threads under determined conditions to prevent the resource overuse. 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 a 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 increasing the memory level parallelism alleviating the resource contention among threads.

In this paper, we explain and discuss how the Runahead mechanism [5][11] can be adapted to SMT processors in order to make Runahead Threads possible. Although both Runahead and SMT processors are well-known micro-architecture techniques, they are two different and clearly separate techniques that have not been considered together before. The novelty of our paper is to join both mechanisms to improve the performance of memory bound threads without prejudicing ILP threads. As far as we know, it has not been previously proposed nor studied at literature. 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 remainder of this paper is organized as follows: Section 2 discusses previous related work. Section 3 describes our experimental environment and simulation tool. We describe in detail in Section 4 how the Runahead mechanism is applied to SMT processors. Later, we analyze some design issues in Section 5. We then evaluate the SMTRA proposal with regard to state-of-the art fetch and resource allocation policies in Section 6. Finally, concluding remarks are given in Section 7.

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 resource conflicts through a 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.

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* [17] and *ICOUNT* [17], 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* [16] 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* [16] 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.

The recent proposal [14], is the work most related to ours. However, unlike our proposal, they execute only some extra instructions indicated by an MLP predictor. After that, they stall or flush the thread. This technique has two limiting factors. Firstly, this technique relies on the MLP predictor accuracy to speculatively execute instructions. Secondly, the number of speculative instructions is limited by the long-latency shift register (LLSR) size. These two factors can reduce the opportunities to improve the performance because it limits the amount of data prefectching by distant MLP.

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 Multicore architectures.

In the context of SMT processors, several approaches (TME [18], SSMT [2] and DDMT [12]) try to exploit the thread level parallelism to speculate and execute various threads simultaneously. Most of these pre-computation 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 or assisted threads*) run ahead of the main thread and trigger cache misses earlier on its behalf, thereby hiding the memory latency. Recent proposals [4][20] 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 and insert special instructions in the code.

Runahead Threads share some aspects with these pre-execution techniques presented at literature. However, these techniques use a different approach. 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. On the contrary, our mechanism does not spawn new threads nor insert specific instructions to switch normal threads into Runahead threads.

Moreover, these pre-execution techniques never focus on controlling SMT resource usage; they only speculate on register and memory values, as well as on control flow between the threads. Our novel approach uses the same thread context to take benefit of long-latency useless and harmful periods. In these periods, we apply Runahead Threads to both exploit the memory level parallelism and alleviate resource contention of memory bounded threads.

3 Experimental Framework

Our simulation environment is based on an SMT execution-driven simulator derived from SMTSIM [15]. 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.

The simulated processor uses a shared reorder buffer among threads. Probably, using a separate ROB per thread would require less hardware complexity, especially to implement the Runahead mechanism. However, using a shared ROB, we want to expose the mechanism to the possible contention of a critical resource such a reorder buffer. The additional complexity mainly lies in making a selective squash of the speculative instructions once the runahead mode finishes. Nevertheless, this process must be implemented in an SMT processor with shared ROB for branch misprediction recovery.

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

Table 1. SMT processor baseline configuration

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 the different benchmarks that compose the workloads, we select 300 millions representative instructions for each benchmark using the reference input set. To identify the most representative simulation segments, we have analyzed the distribution of basic blocks using Sim-Point [13]. In the case of the multithreaded workloads, the finalization model consists on stopping the execution when any thread had committed 300 million instructions.

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

To create the multithreaded workloads, we consider only groups with 2 and 4 threads, since several studies [8][6] have shown that SMT performance saturates or even degrades for workloads with more than 4 contexts. We made benchmarks characterization based on the L2 cache miss rate of each program simulated in a single-threaded context. Next, we group them and distinguish three types of workloads: high instruction level parallelism threads (ILP), memory-bounded threads (MEM) and a mixture of both (MIX). Table 2 shows our simulation workload sets identified by the number of threads it contain and the type of these threads. We note that we chose groups of workloads large enough to avoid results deviation due to a particular workload specific behaviour [1].

4 Runahead in SMT Processors

Runahead execution is a well-known mechanism whose goal is bringing speculatively 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 mechanism consists of avoiding the blockage of the instruction window due to long-latency operation (eg. a load that misses in the L2-cache). Instead, the processor continues executing in a speculative instructions, trying to follow the more likely program path. Runahead benefit comes from pre-execution of these speculative instructions that issue future memory operations improving prefetch efficiency for the data and instruction caches.

4.1 Runahead Operation Background

Runahead mechanism basically prevents the reorder buffer from stalling on long-latency memory operations by executing speculative instructions. 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 register and memory 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 speculative execution would have generated useful data and instruction prefetches, improving the behaviour of the memory hierarchy during the real execution.

4.2 Runahead Threads

Resource conflicts are an important problem for SMT processor performance. 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

[1] In order to make equivalent comparisons of our proposal, we mainly try to select group of applications referenced in earlier researches to create these workloads (ILP, MIX or MEM)

resources during short time. The invalid instructions use 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.

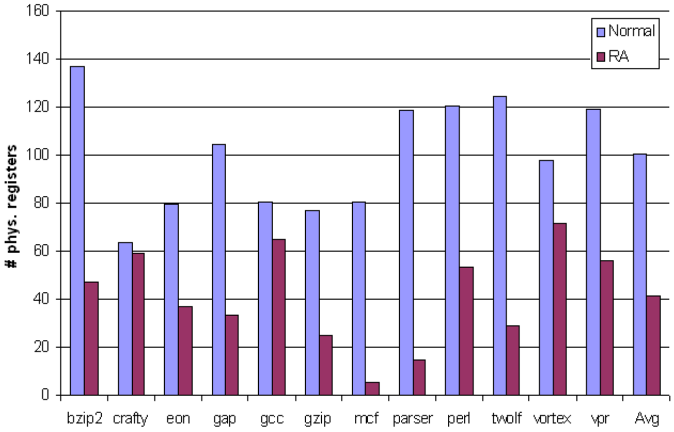


Figure 1. Average physical register used per cycle between the operation modes in RA for SPECINT 2000

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 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.

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 $192 = 320 - (32x4)$ rename registers when 4 threads are running. To correctly recover the architectural state, each thread only needs to checkpoint the contents 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 a candidate to be freed and to be used soon for the rest of threads.

Mutlu *et al.* [11] introduce the runahead cache to provide communication of data and invalid status between runahead loads and stores. This additional cache holds the results and INV status of the pseudo-retired stores. Basing 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 incongruous data due to runahead cache block replacements.

From the SMT point of view, using a runahead cache can be expensive in term of hardware. The RA cache needs to be larger to avoid a mayor pressure on line contention among threads. Likewise, it is necessary to include a new identification tag for each thread to distinguish the block owners. We measure the performance with and without the runahead cache for considering the need to include it in an SMTRA environment. The results show that runahead cache use does not modify the performance with regard to SMTRA without this cache for our multithreaded workloads [2]. Then, we based on this result and the fact that a runahead cache implies use more area in the SMT core to this structure to decide not to use it in our runahead SMT model. The functionality difference is that some loads dependent on previous retired stores use stale values to do speculative memory accesses.

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 Additional Design Analysis

Now, we discuss and analyze relevant design aspects evaluation related with Runahead mechanism in the SMT processors environment.

[2] In [11], the performance deviation without RA cache in SPEC2000 is also very small for an out-of-order processor

5.1 Saving resources without FP Operations

The Runahead mechanism improves performance mainly due to the pre-execution of memory operations and the corresponding prefetching. 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. 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 [10] as an efficient optimization for runahead execution in 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 does not need the floating-point resources of the SMT processor for the rest of non-speculative threads. So, once an instruction is detected as an FP operation in the decode stage, it is directly invalidated and proceeds directly to pseudo-commit. With this modification, FP instructions do not occupy any processor resources in runahead mode after they are decoded. Therefore, the FP issue queue, the FP functional units and the FP physical register file are not used by FP runahead instructions. The exception are the FP loads and stores which are treated as prefetch instructions since their addresses are integer computations.

To examine the performance impact of eliminating the FP operation during runahead mode, we simulate the different workloads in SMT processor that do not execute FP instructions during runahead mode versus one that does it. The slight performance changes were negligible, so we decide to not execute floating-point instructions in runahead threads.

5.2 Thread Priority

With the introduction of Runahead execution in the SMT architecture context, 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 good scheme to manage this new situation. We recall that the ICOUNT scheme only takes into account the amount of instructions in the pre-execute stages to calculate the thread priority, independently of the kind of thread (speculative or not). In order to analyze this new environment, we investigate several modification of the ICOUNT algorithm varying the thread priorities.

One of the proposals evaluated consist of giving high priority to normal threads against runahead ones at the fetch stage. Therefore, normal threads have first the opportunity to get into the pipeline although, following the same conditions that standard ICOUNT imposes among them. Subsequently, the runahead threads take profit of the possible remaining fetch slots after scheduling the normal threads. We also probe another scheme in combination with this previous one, adding a new level of priority at the issue queues with the same criteria. In this case, the advantage for normal threads is not only at the fetch stage but also at the moment of taking the functional units in the issue stage. Finally, the last fetch priority policy scheme we analyze consists of inverting the rights, giving priority to runahead threads opposite to the normal ones.

Figure 2 shows the different performance obtained by the ICOUNT standard fetch policy and the thread priority proposed schemes described above. The four bars of each workload set respectively represent the harmonic mean of throughput for ICOUNT base policy, fetch priority to normal thread (FPNT), both fetch and issue priority to normal threads (BPNT) and fetch priority to runahead threads (FPRT).

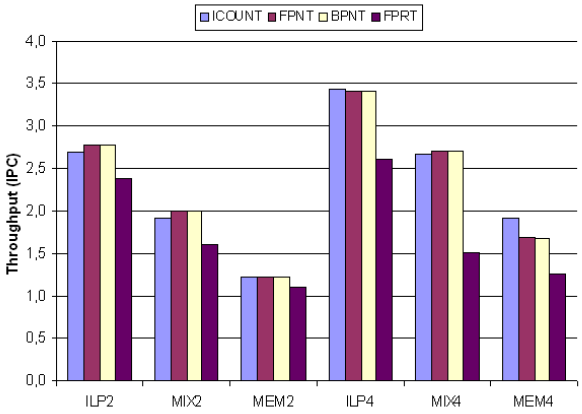


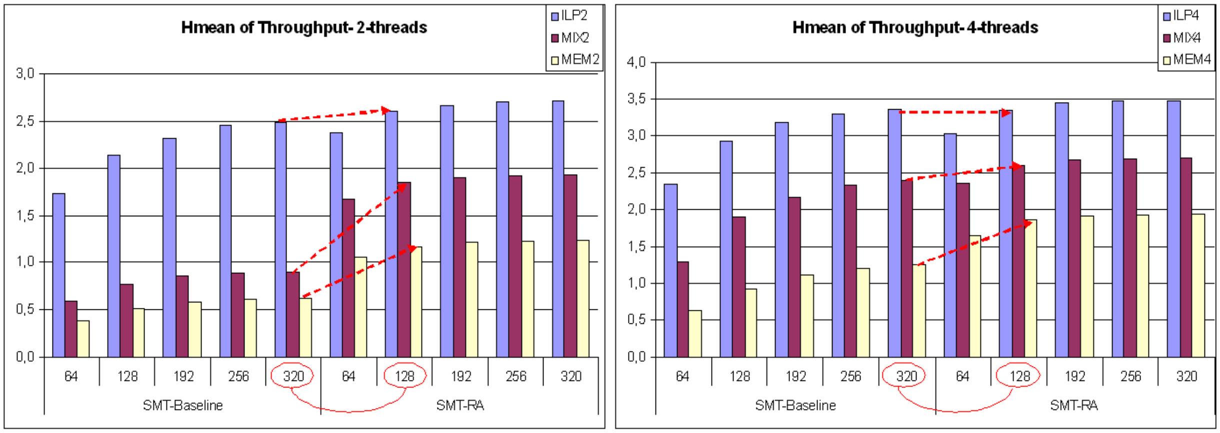
Figure 2. Thread priority scheme analysis for SMTRA

Figure 2 shows that giving priority to Runahead threads (FPRT) is a bad choice since the performance considerably decreases. This policy favours speculative Runahead threads while the non-speculative threads, which do useful work, are being delayed. On the other hand, both FPNT and BPNT get the same performance level. We analyze our results and they show very little differences in the number of functional unit conflicts between using FPNT or BPNT, which indicates that the issue stage priority is not relevant in this case. Therefore, we discard the BPNT scheme due to its extra complexity to select priority instructions from the issue queues to get equal performance as FPNT.

High fetch priority to normal threads (FPNT) seems a good option, especially for ILP and MIX workloads. This scheme favours the threads that have long periods of high-level parallelism, since the instruction streams flow quickly in the pipeline, having fewer instructions in the front-end stages. However, this effect harms memory-bounded threads due to impede the speculative runahead instructions to do prefetching (see MEM workloads). Therefore, 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.

5.3 Register File size Impact

The register file is an important shared resource inside an SMT processor. One of the key issues in SMT design is the size of the physical register file. This parameter, related with the ISA, sets the number of threads that an SMT processor is able to simultaneously execute in its core. With N architectural registers ISA, N physical registers are needed and reserved to keep the precise state for each thread. For example, in a 4-threaded SMT with an Alpha ISA (like our model) $32x4 = 128$ physical registers are needed to maintain the architectural state of each thread in both integer and floating point register files. The rest of extra physical registers are available to share between threads for renaming purposes. Then, the number of renaming registers must be high enough to support several simultaneous threads. However, higher register files produce larger access delays and more complex designs.



(a) 2-thread Workloads

(b) 4-thread Workloads

Figure 3. Hmean of Throughput (IPC) relative to workloads for different register file size.

We examine what happens with throughput for SMT with the ICOUNT fetch policy (baseline) versus SMT with runahead threads while the register file size is varied. In figures 3(a) and 3(b) we show the harmonic mean of throughput for the 2-thread and 4-thread workloads respectively as a function of the register file size, in which the number of physical registers for renaming are varied from 64 to 320 (for both integer and fp). As we can observe, the throughput lessens as the number of registers diminishes, especially in the case of 4 threads. However, this reduction is less slanting when runahead mechanism is used than the baseline. For instance, the MEM4 workloads in the baseline suffer from 50% slowdown when pass from 320 registers to 64 while using runahead the slowdown is 15%. Therefore, an SMT processor with runahead execution is less sensitive to register file size constraints than without it.

It is interesting to note that SMTRA makes possible the use of smaller register files without degrading performance over the SMT baseline. If we compare the throughput of the SMT-baseline for all configurations versus the SMTRA with 64 registers case, the latter overcomes the former for almost all combinations of cases except for the ILP workloads. There is a sample that remarks this valuable result: the performance of SMTRA using 128 renaming physical registers impressively speeds up the performance of SMT baseline using 320 physical registers, that is, using much less than half of the registers that the other. The performance gain is corresponded with 4.5%, 107% and 87% for ILP, MIX and MEM 2-thread workloads and 0.2%, 8.5% and 48% for ILP, MIX and MEM 4-thread workloads respectively. Therefore, we can extract an important key design from these results: SMTRA allows using much smaller register files without degrading performance.

6 Comparative Evaluation

In this section, we evaluate the performance and fairness of the 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:

$$Average\_IPC = \frac{\sum_{i=1}^{n} IPC_{MT,i}}{n} \tag{1}$$

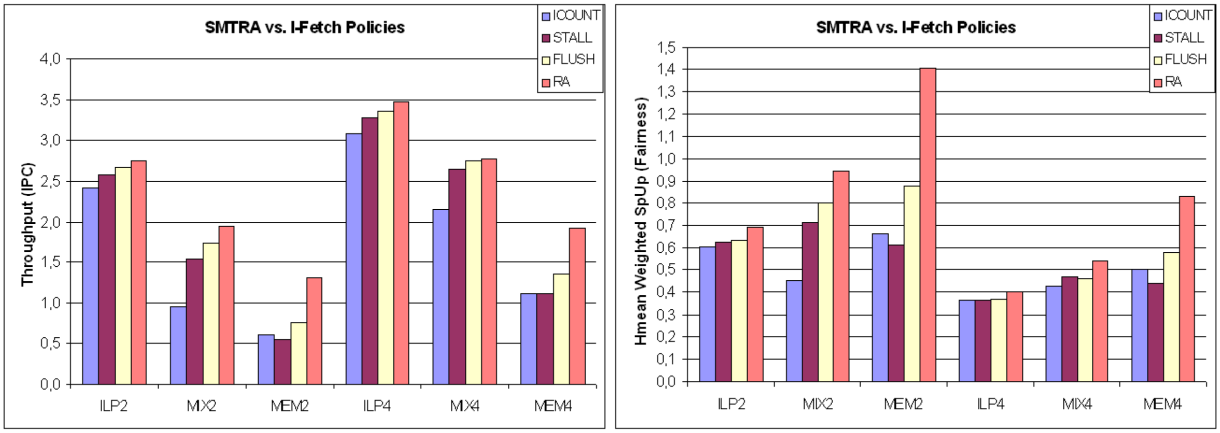
The other metric represents the fairness-performance balance, proposed in [9], which is the harmonic mean of IPC speedup of each thread compared to single-threaded execution:

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

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.

6.1 SMTRA versus 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* proposed by [16], together with SMTRA. All of them are implemented over the ICOUNT policy, which we use as the simple baseline reference.



(a) Average Throughput (IPC)

(b) Hmean Weighted SpeedUp

Figure 4. Throughput and Hmean relative to workloads for different I-Fetch policies.

Figures 4 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 4(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 causes this technique to fall into less stalls or flushs than the others, and then it does not slow down the program progress.

Figure 4(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.

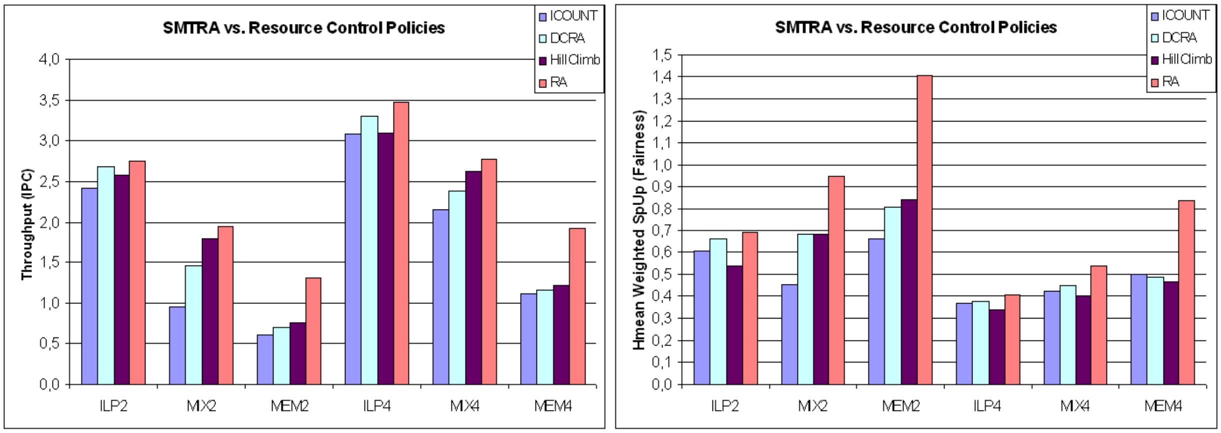
6.2 SMTRA versus Resource Control Policies

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). Regarding HillClimbing, we use the performance function based on the throughput (named Hill-Thru in [3]). The other two possibilities for performance function (with weighted speedup and harmonic mean) use the IPC of each benchmark in single thread as an external input. In these sense, we consider that these options for HillClimbing are highly dependent of variability of program parameters. They require repeating the single program execution per different input for each thread to execute in a real multithreaded processor to guide the mechanism.

Figure 5 shows the IPC throughput (a) and harmonic mean (b) for ICOUNT (baseline), DCRA, HillClimbing and SMTRA respectively. As Figure 5(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 5(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



(a) Average Throughput (IPC)

(b) Hmean Weighted SpeedUp

Figure 5. Throughput and Hmean relative to workloads for different resource control policies.

score for MEM workloads being 73% better than DCRA and 72% better than HillClimbing.

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 techniques evaluated in this subsection (both DCRA and HillClimbing) 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 resource 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.

6.3 Efficiency: Performance vs. Energy

Unlike previous resource control techniques, Runahead Threads involves speculative execution that it is translated in a higher number of executed instructions. Flush is the only other technique that executes additional instructions, since it executes 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.

Finally, in this section we show a power consumption discussion about the efficiency related to the performance gain achieved and the additional energy consumed. To measure this efficiency, we use the commonly accepted Energy-Delay2 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 kind of instruction executed, we assume that all the instructions consume the same amount of energy to simplify our analysis. On the other hand, the delay is counted as the average CPI. Therefore, the final resulting formula is:

$ED^2 = Num\_Executed\_Instr.xCPI^2$

This formula provides an approximation of how efficiently the instructions are executed, regarding energy and consumption. In figure 6 we show the ED^2 for the SMT processor with ICOUNT (left-most bar), Flush and SMTRA (rigth-most bar) for each group of workloads. The bars are normalized to the ICOUNT values. So, the higher the bar is, the more energy is wasted per executed instruction with regard to ICOUNT, and vice versa.

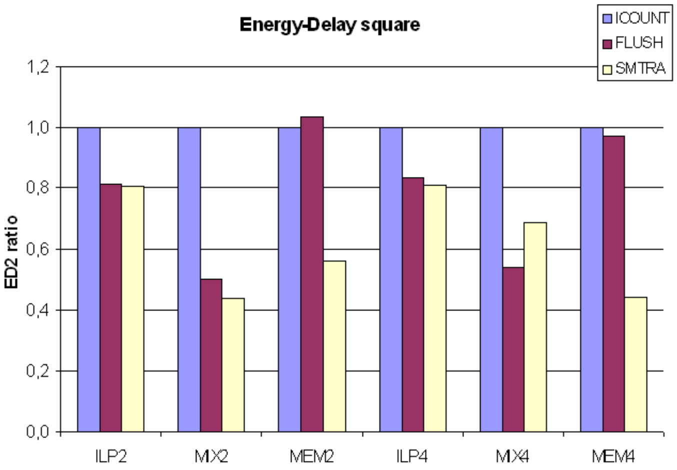


Figure 6. Energy-Delay2

As shown in Figure 6, SMTRA is the most efficient teechnique in terms of ED^2 for the evaluated mechanisms. SMTRA has 0,6 ED2 for 2-thread workloads and 0,78 ED2 for 4-thread workloads on average with regard to ICOUNT, while Flush presents 0,78 ED2 for both same workloads. Then, except for MIX4, the SMT processor consumes less energy per committed instruction using SMTRA that without it to get better performance. For instance, SMTRA is much more efficient for MEM workloads, which is considerably lower than Flush and ICOUNT.

7 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. Additionally, SMTRA presents a good ratio of power consumption regarding its performance improvement.

This work opens new ways to use Runahead Threads or similar speculative mechanisms as an alternative to resource contention in SMT processors. SMTRA is also orthogonal to some dynamic resource scheduling schemes and it is possible to combine them to achieve better resource aware schemes. In addition, without forgetting the power consumption, there are still some interesting trade-offs on SMTRA to create more efficient schemes. This is a topic which we were working on.

References

[1] Francisco J. Cazorla, Alex Ramirez, Enrique Fernandez, and Mateo Valero. Dynamically controlled resource allocation in smt processors. In *MICRO-37*, pages 171–182, 2004.

[2] Robert S. Chappell, Jared Stark, Steven K. Reinhardt, Yale N. Patt, and Sangwook P. Kim. Simultaneous subordinate microthreading (ssmt). *ISCA-26*, 00, 1999.

[3] Seungryul Choi and Donald Yeung. Learning-based smt processor resource distribution via hill-climbing. In *ISCA-33*, pages 239–251, Washington, DC, USA, 2006.

[4] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic speculative precomputation. *MICRO'01*, page 306, 2001.

[5] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS-11*, pages 68–75, New York, NY, USA, 1997.

[6] Ronaldo Gonalves, Eduard Ayguad, Mateo Valero, and Philippe 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] Sébastien Hily and André 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 *IPASS*, 2001.

[10] Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Techniques for efficient processing in runahead execution engines. In *ISCA-32*, pages 370–381, 2005.

[11] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA'03*, page 129, Washington, DC, USA, 2003.

[12] Amir Roth and Gurindar S. Sohi. Speculative data-driven multithreading. *HPCA'01*, 2001.

[13] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT'01*, pages 3–14, Washington, DC, USA, 2001.

[14] Everman Stijn and Eeckhout Lieven. A memory-level parallelism aware fetch policy for smt processors. In *HPCA'07*, pages 240–249, 2007.

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

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

[17] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *ISCA-23*, pages 191–202, 1996.

[18] Steven Wallace, Brad Calder, and Dean M. Tullsen. Threaded multiple path execution. In *ISCA-25*, pages 238–249, Washington, DC, USA, 1998.

[19] Wayne Yamamoto and Mario Nemirovsky. Increasing superscalar performance through multistreaming. In *PACT'95*, pages 49–58, Manchester, UK, 1995.

[20] Weifeng Zhang, Dean M. Tullsen, and Brad Calder. Accelerating and adapting precomputation threads for efficient prefetching. In *HPCA'13*, Phoenix, AZ, USA, 2007.