Dynamically Controlled Resource Allocation in SMT Processors

Francisco J. Cazorla, Alex Ramirez, Mateo Valero
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Jordi Girona 1-3, D6
08034 Barcelona, Spain
{fcazorla, aramirez, mateo}@ac.upc.es

Enrique Fernández
Universidad de Las Palmas de Gran Canaria
Edificio de Informática y Matemáticas
Campus Universitario de Tafira
35017 Las Palmas de G.C., Spain
efernandez@dis.ulpgc.es

Abstract

SMT processors increase performance by executing instructions from several threads simultaneously. These threads use the resources of the processor better by sharing them but, at the same time, threads are competing for these resources. The way critical resources are distributed among threads determines the final performance. Currently, processor resources are distributed among threads as determined by the fetch policy that decides which threads enter the processor to compete for resources. However, current fetch policies only use indirect indicators of resource usage in their decision, which can lead to resource monopolization by a single thread or to resource waste when no thread can use them. Both situations can harm performance and happen, for example, after an L2 cache miss.

In this paper, we introduce the concept of dynamic resource control in SMT processors. Using this concept, we propose a novel resource allocation policy for SMT processors. This policy directly monitors the usage of resources by each thread and guarantees that all threads get their fair share of the critical shared resources, avoiding monopolization. We also define a mechanism to allow a thread to borrow resources from another thread if that thread does not require them, thereby reducing resource under-use. Simulation results show that our dynamic resource allocation policy outperforms a static resource allocation policy by 8%, on average. It also improves the best dynamic resource-conscious fetch policies like FLUSH++ by 4%, on average, using the harmonic mean as a metric. This indicates that our policy does not obtain the ILP boost by unfairly running high ILP threads over slow memory-bounded threads. Instead, it achieves a better throughput-fairness balance.

1 Introduction

Superscalar processors increase performance by exploiting instruction level parallelism (ILP) within a single application. However, data and control dependences reduce the ILP of applications. As a result, when the available ILP is not high enough, many processor resources remain idle and do not contribute to performance. Simultaneous multithreaded (SMT) processors execute instructions from multiple threads at the same time, so that the combined ILP of multiple threads allows a higher usage of resources, increasing performance [7][15][16][17]. However, threads not only share the resources, they also compete for them.

In an SMT, resource distribution among threads determines not only the final processor performance, but also the performance of individual threads. If a single thread monopolizes most of the resources, it will run almost at its full speed, but the other threads will suffer a slowdown due to resource starvation. The design target of an SMT processor determines how the resources should be shared. If increasing IPC (throughput) is the only target, then resources should be allocated to the faster threads, disregarding the performance impact on other threads. However, current SMT processors are perceived by the Operating System (OS) as multiple independent processors. As a result, the OS schedules threads onto what it regards as processing units operating in parallel and if some threads are favored above others, the job scheduling of the OS could severely suffer. Therefore, ensuring that all threads are treated fairly is also a desirable objective for an SMT processor that can not be quickly disregarded.

In current SMTs, resource distribution among threads is either static or fully dynamic. A static resource distribution (used, for example, in the Pentium 4) evenly splits the resources among the running threads. This ensures that no single thread monopolizes the resources and that all threads are treated equally. This scheme suffers the same problem as a superscalar processor: if any thread does not fully use

the allocated resources, these are wasted and do not contribute to performance. Dynamic sharing of resources is accomplished by running all threads in a common resource pool and allowing threads to freely compete for them. In a dynamically shared environment, it is the fetch policy (I-fetch) that actually controls how resources are shared. The fetch policy determines which threads can enter the processor to get the opportunity of using available resources. However, current fetch policies do not exercise direct control over how resources are distributed among threads, using *only* indirect indicators of potential resource abuse by a given thread, like L2 cache misses. Because no direct control over resources is exercised, it is still possible that a thread will obtain most of the processor resources, causing other threads to stall. Also, to make things worse, it is a common situation that the thread which has been allocated most of the resources will not release them for a long period of time. There have been fetch policies proposed [2][9] that try to detect this situation, in order to prevent it by stalling the thread before it is too late, or even to correct the situation by squashing the offending thread to make its resources available to other threads [14], with varying degrees of success. The main problem of these policies is that in their attempt to prevent resource monopolization, they introduce resource under-use, because they can prevent a thread from using resources that no other thread requires.

In this paper, we show that the performance of an SMT processor can significantly be improved if a direct control of resource allocation is exercised. On the one hand, at any given time, 'resource hungry' threads must be forced to use a limited amount of resources. Otherwise, they could monopolize shared resources. On the other hand, in order to allow 'resource hungry' threads to exploit ILP much better, we should allow them to use as many resources as possible while these resources are not required by the other threads. This is the trade-off addressed in this paper.

In order to control the amount of resources given to each thread, we introduce the concept of a *resource allocation policy*. A resource allocation policy controls the fetch slots, as instruction fetch policies do, but in addition it exercises a direct control over *all* shared resources. This direct control allows a better use of resources, reducing resource under-utilization. The main idea behind a smart resource allocation policy is that each program has different resource demands. Moreover, a given program has different resource demands during the course of its execution. We show that the better we identify these demands and adapt resource allocation to them, the higher the performance of the SMT processor gets.

In this paper, we propose such a resource allocation policy called Dynamically Controlled Resource Allocation (DCRA). DCRA first classifies threads according to the amount of resources they require. This classification provides DCRA with a view of the demand that threads have of each resource. Next, based on the previous classification, DCRA determines how each resource should be distributed among threads. Finally, each cycle DCRA directly monitors resource usage, without relying entirely on indirect indicators. Hence, it immediately detects that a thread is exceeding its assigned allocation and stalls that thread until it no longer exceeds its allocation. Our results show that our DCRA policy outperforms a static resource allocation policy(SRA) [4][11][12] and also the best dynamic resource-conscious fetch policies like FLUSH++ [1] in both throughput and fairness [10]. Throughput results show that DCRA improves SRA by 7% and FLUSH++ by 1%, on average. Fairness results, using the harmonic mean as a metric, indicate that DCRA outperforms SRA by 8% and FLUSH++ by 4%, on average. Both results confirm that DCRA achieves better throughput than the other policies and in addition presents a better throughput-fairness balance than them.

The remainder of this paper is structured as follows: we present related work in Section 2. In Section 3 we present our new policy. In section 4, we explain the experimental environment. Sections 5 presents the simulation results. Conclusions are given in Section 6.

2 Related work

Current SMT processor proposals use either static resource allocation or fully flexible resource distribution. The static sharing model [4][11][12] evenly splits critical resources (mainly registers and issue queues) among all threads, ensuring that any thread monopolizes a resource, causing other threads to wait for that resource. However, this method lacks flexibility and can cause resources to remain idle when one thread has no need for them, while the other threads could benefit from additional resources.

An alternative to static partitioning of resources is to have a common pool of resources that is shared among all threads. In this environment, the fetch policy determines how resources are shared, as it decides which threads enter the processor and which are left out.

ROUND-ROBIN [15] is the most basic form of fetch and simply fetches instructions from all threads alternatively, disregarding the resource use of each thread.

ICOUNT [15] prioritizes threads with few instructions in the pre-issue stages and presents good results for threads with high ILP. However, an SMT has difficulties with threads with high L2 miss rate. When this situation happens, ICOUNT does not realize that a thread can be blocked and does not make progress for many cycles. As a result, shared resources can be monopolized for a long time.

STALL [14] is built on top of ICOUNT to avoid the problems caused by threads with a high cache miss rate. It de-

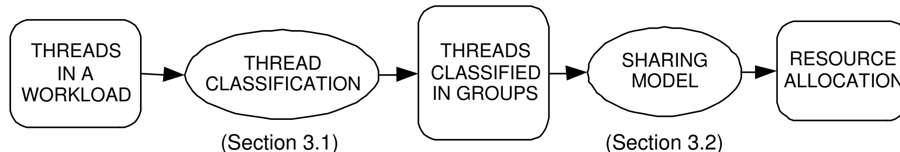


Figure 1. Main tasks of the DCRA policy

tects that a thread has a pending L2 miss and prevents the thread from fetching further instructions to avoid resource abuse. However, L2 miss detection already may be too late to prevent a thread from occupying most of the available resources. Furthermore, it is possible that the resources allocated to a thread are not required by any other thread, and so the thread could very well continue fetching instead of stalling, producing resource under-use.

FLUSH [14] is an extension of STALL that tries to correct the case in which an L2 miss is detected too late by deallocating all the resources of the offending thread, making them available to the other executing threads. However, it is still possible that the missing thread is being punished without reason, as the deallocated resources may not be used (or fully used) by the other threads. Furthermore, by flushing all instructions from the missing thread, a vast amount of extra fetch and power is required to redo the work for that thread.

FLUSH++ [1] based on the idea that STALL performs better than FLUSH for workloads that do not put a high pressure on resources, that is, workloads with few threads that have high L2 miss rate. Conversely, FLUSH performs better when a workload has threads that often miss in the L2 cache, and hence the pressure on the resources is high. FLUSH++ combines FLUSH and STALL: it uses cache behavior of threads to switch among FLUSH and STALL in order to provide better performance.

Data Gating (DG) [2] attempts to reduce the effects of loads missing in the L1 data cache by stalling threads on each L1 data miss. However, when a L1 miss does not cause an L2 miss there is not resource abuse. We have measured that for memory bounded threads less than 50% of L1 misses cause an L2 miss. Thus, to stall a thread every time it experiences an L1 miss may be too severe.

Predictive Data Gating (PDG) [2] and DC-PRED [9] work like STALL, that is, they prevent a thread from fetching instructions as soon as a cache miss is predicted. By using a miss predictor, they avoid detecting the cache miss too late, but they introduce yet another level of speculation in the processor and may still be saving resources that no other thread will use. Furthermore, cache misses prove to be hard to predict accurately [18], reducing the advantage of these techniques.

Recently, IBM has presented the Power5 [8] that has 2 cores where each core is a 2-context SMT. However, no information about resource assignment inside a core has been released.

Our technique, DCRA, first dynamically determines the amount of resources required by each thread and prevents threads from using more resources than they are entitled to use. We continuously monitor the processor and redistribute resources as threads change from one phase to another, or when they do not require resources that they needed before. In the following sections we describe how our mechanism determines which threads are resource-hungry, which require few resources, and the resource allocation model to distribute resources among all threads.

3 The DCRA policy

To perform an efficient resource allocation, it is necessary to take into account the different execution phases of a thread. Most threads have different behavior patterns during their execution: they alternate high ILP phases and memory-bounded phases with few parallelism, and thus their resource needs change dynamically. We must take into account this varying behavior in order to allocate resources where they will be best used, and also to allocate resources where they are needed most.

Figure 1 shows a diagram of how resource allocation policies like DCRA, work. DCRA first dynamically classifies threads based on the execution phase they are in, high ILP or memory-bounded (Section 3.1.1). Next, we determine which resources are being used by each thread in the phase it is in (Section 3.1.2). After that, DCRA uses a sharing model to allocate resources to threads based on the classification previously made (Section 3.2). Finally, the sharing model also ensures that threads do not exceed their allocated resources.

Our model bases on the fact that threads without outstanding L2 misses require less resources to exploit ILP than threads with L2 misses. Figure 2 shows the average IPC as we vary the amount of resources given to SPEC 2000 benchmarks when executed in single-thread mode and the data L1 cache is perfect [1]. For this experiment we use 160

[1] the average results for the FP registers and issue queue were obtained only from FP benchmarks

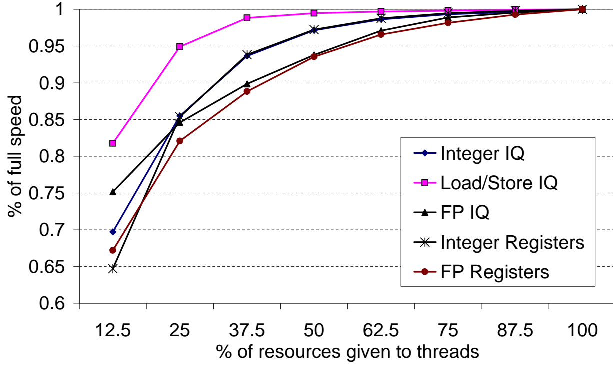


Figure 2. Average IPC of SPEC benchmarks as we vary the amount given to them when the data L1 cache is perfect.

rename registers, 32-entry issue queues, and the remainder parameters of our baseline configuration shown in Section 4. For example, the point 25% for the integer IQ shows the average IPC when benchmarks are allowed to use 25% of the integer IQ and all other resources. In general we see that with few resources threads run at almost the same speed than when they use all the resources of the machine (full speed). We see that only with 37.5% of resources (12 IQ entries and 60 physical registers) threads run at approximately 90% of their full speed.

Our objective is to give additional resources to memory-bound threads as these resources are clearly not needed by threads without outstanding cache misses. By giving more resources to missing threads we obtain benefits as we give the out-of-order mechanism more opportunity to overlap multiple L2 misses, increasing the memory parallelism of the application without really harming the performance of the remaining threads.

3.1 Thread classification

DCRA classifies threads based on how many resources they need to efficiently exploit ILP depending on the phase of the thread. Moreover, each thread is classified depending on which critical resources it actually uses and which resources it does not need. The DCRA classification is continuously re-evaluated to adapt to the changing behavior of threads. Hence, it can dynamically adapt the resource allocation to the specific needs of the running threads.

DCRA bases on the idea that resources for a thread will be allocated according to both classifications, thread phase and critical resources needed. On the one hand, threads in a memory-bounded phase with difficulties to exploit ILP will borrow resources from faster threads. On the other hand, critical resources will only be distributed among those threads which actually can use them.

3.1.1 Thread phase classification

It is important to note two points about the thread classification made by DCRA. First, we do not classify a thread for its entire lifetime: we distinguish the different phases in a thread's execution, adapting to the dynamic changes in resource requirements. Second, our policy does not need to know the exact amount of each resource that a thread needs. We only classify threads into those requiring few resources to achieve high-performance, and those with higher requirements, so that one thread group can temporarily give additional resources to the other group.

We classify threads in two groups: the $Fast$ group, and the $Slow$ group. We use cache behavior to determine in which group to place a thread. When a thread experiences a cache miss, it runs much slower than it could and it holds resources that will not be released for a potentially long time: until the missing load is committed, each instruction holds a reorder buffer (ROB) entry and, many of them, a physical register. Also, all instructions depending on the missing load hold an instruction queue (IQ) entry without making any progress as long as the offending load is not resolved. On the other hand, threads which do not experience cache misses are able to exploit ILP with few resources. Please, note that they still require IQ entries and physical registers, but they release these resources shortly after allocating them, so they are able to run on a reduced set of resources.

After having explored several possibilities, we classify threads based on L1 data cache misses. Threads with pending L1 data misses are classified in the slow group, because they may allocate resources for a long period of time, and threads with no pending L1 data cache misses are classified in the fast group, because they will be able to run on a rapidly cycling set of resources.

3.1.2 Resource usage classification

Given the classification described above, we could already distribute resources among threads taking into account which ones require additional resources and which ones can do with less than their equal share. However, not all threads use every available resource in the processor during their entire lifetime and assigning them resources of a type that is not required would effectively be wasting these resources. For that reason, we also classify each thread as *active* or *inactive* with regard to several processor resources. We proceed as follows: every time a thread uses a given resource, it is classified as *active* for the following Y cycles. If the thread did not use this resource after the assigned Y cycles, the thread is classified as *inactive* until it uses that type of resource again.

If a thread is classified as inactive for a given resource, then we assume that it does not compete for the resource and its share can be evenly split among the remaining competing threads. In our setup this method is effective for the floating point resources when a thread is in an integer computation phase. In other SMT configurations using other types of resources, e.g., vector resources [3], this method would also be effective. Note that the activity classification is associated to a specific type of resource and that a thread can be active with respect to a certain resource and inactive with respect to others.

The thread phase classification and the resource usage classification are orthogonal. Hence, for each resource we have 4 possible classifications for a thread: fast-active (F_A), fast-inactive (F_I), slow-active (S_A), and slow-inactive (S_I). The main characteristics of this classification are that inactive threads (X_I) do not use their share of a given resource and that slow threads (S_x) require more resources to exploit ILP than fast threads (F_x) do.

3.2 The sharing model

The sharing model determines how shared resources are distributed among threads. A key point in any sharing model is to be aware of the requirements of each group of threads: the more accurate the information, the better the sharing.

Our sharing model starts from the assumption that all threads receive an equal share of each shared resource. On average, each thread gets E entries of each resource, given in the equation (1), where R is the total number of entries of that resource and T is the number of running threads.

$$E = \frac{R}{T} \quad (1)$$

Next, we take into account that slow threads require more resources than fast threads. Hence, fast threads can share part of their resources with slow threads. This way, slow threads are assigned their equal share and also borrow some additional resources from those threads that can do without them. This introduces a *sharing factor*, C, that determines the amount of resources that fast threads give to each slow thread. The value of C depends on the number of threads: if there are few threads, then there is little pressure on resources and many resources are assigned to each thread. Hence, fast threads have more resources to lend out. We have tested several values for this sharing factor and $C = \frac{1}{T+4}$ gives the best results for low memory latencies. With this sharing model, slow threads increase their share with the resources given to them by fast threads. Hence each of the slow threads is entitled to use at most E_{slow} entries, as shown in equation (2), where F is the number of fast threads.

$$E_{slow} = \frac{R}{T}(1 + C * F) \quad (2)$$

At this point, our sharing model takes into account which threads require more resources and which threads can give part of their share to these resource-hungry threads. However, we still do not account for the fact that not all threads use every type of resource in the processor. To account for this information, we use a separate resource allocation for each type of resource and take into account that threads which are inactive for a certain resource (those in the S_I and F_I groups) can give their entire share of that resource to the other threads. Hence, each active thread has $E = \frac{R}{F_A+S_A}$ reserved entries of a resource, since inactive threads do not compete for them. Moreover, we have to consider that fast active threads also share a part of their resources with slow active threads, as determined by the sharing factor C, which we re-define as $C = \frac{1}{F_A+S_A}$. Hence, the number of entries that each slow active thread is entitled to use is re-defined as:

$$E_{slow} = \frac{R}{F_A + S_A}(1 + C * F_A) \quad (3)$$

This final model distributes resources only among active threads, those actually competing for them, and gives more resources to threads in the slow phases, taking them from the threads in high ILP phases.

entry	F_A	S_A	E_{slow}
1	0	1	32
2	1	1	24
3	0	2	16
4	2	1	18
5	1	2	14
6	0	3	11
7	3	1	14
8	2	2	12
9	1	3	10
10	0	4	8

Table 1. Pre-calculated resource allocation values for a 32-entry resource on a 4-thread processor. F_A and S_A denote the number of fast and slow active threads respectively

Example. Assume a shared resource with 32 available entries in a processor that runs 4 threads. Table 1 shows the resource allocation of slow active threads for all cases in this example situation. In case that all threads are in a slow phase and active for that resource (table entry 10), they would receive 8 entries each. In case where 3 threads are in the fast group and 1 is in the slow group, all active for the resource (table entry 7), the slow thread would be allocated 14 entries, leaving 18 entries for the fast threads. In case where 3 threads are in the fast group (1 is inactive for the resource and 2 are active), and 1 is in the slow group (table entry 4), the slow thread would be allocated 18 entries, and the fast active threads would be left with 14 entries. The

inactive fast thread does not allocate any entries for this resource. The other entries are computed in the same way.

3.3 Resource allocation policies vs. Instruction fetch policies

The main differences between a dynamic allocation policies, like DCRA, and an I-fetch policy are the following: the *input information* involved and the *response action*.

- The input information is the information used by the policy to make decisions about resource assignment. Usually, this information consists of indirect indicators of resource use, like L1 data misses or L2 data misses.

- The response action is the behavior of a policy to control threads. For example, this response action could be to stall the fetch of a thread.

I-fetch policies just control the fetch bandwidth. All I-fetch policies we have seen, except ICOUNT and ROUND-ROBIN, stall the fetch of threads. FLUSH, in addition, squashes all instructions of the offending thread after a missing load. As input information, I-fetch policies use indirect indicators, like L1 misses or L2 misses, as we have seen in the Related Work Section.

Allocation policies control fetch bandwidth, as I-fetch policies do. As shown in [15], the fetch bandwidth is a key parameter for SMT processors. Hence, control of this resource is essential. In addition, an allocation policies controls *all* shared resources in an SMT processor, since the monopolization of any of these resources causes a stall of the entire pipeline. As input information, allocation policies uses indirect indicators and, in addition, information about the demand and availability of resources. The more accurate the information, the better the resource allocation.

The key point is that I-fetch policies are not aware of the resource needs of threads. They just assume that resource abuse happens when an indirect indicator is activated. That is, indicators are perceived as abuse indicators and, as a consequence, when any of them is activated, the I-fetch policy immediately stalls or flushes a thread. An allocation policy perceives indirect indicators as information on resource demand. As a consequence, it does not immediately take measures on threads with high resource demands. Instead, it computes the overall demand as well as the availability of resources. Then it splits shared resources and fetch bandwidth between threads based on this information. Notice that the objective of our policy is to help, *if possible*, threads in slow phases, i.e., those threads experiencing cache misses. In contrast, previously proposed fetch policies proceed the other way around by stalling/flushing those threads experiencing cache misses.

Other important information to take into account is the number of running threads, because this number determines the pressure on resources. The higher the number of threads, the higher the pressure. Current I-fetch policies do not take this information into account and, as a result, the response action they take may be inadequate. Allocation policies use this information when sharing resources between threads and hence the resource allocation complies with the demand for resources.

3.4 Implementation of the allocation policy

Figure 3 shows the processor modifications required for a possible implementation of our dynamic allocation policy. The modifications focus on two main points:

First, DCRA requires 8 counters per thread (7 resource usage counters and one additional counter to track pending L1 data misses). Like ICOUNT, DCRA tracks the number of instructions in the IQs, but distinguishing each of the three IQ types: integer, fp, and load/store. DCRA also tracks physical registers (integer and fp) and hence 2 more counters are required. As shown below, these two counters are incremented in the decode stage and decremented when the instructions commit. Hence, DCRA does not affect the register file design. To detect inactive threads we maintain an *activity* counter for each floating point resource: fp issue queue and fp physical registers. Finally, like DG and PDG, DCRA keeps track of L1 data misses. The additional complexity required to introduce these counters depends on the particular implementation, but we do not expect it to be more complex than other hardware counters already present in most architectures. Resource usage counters are incremented in the decode stage (indicated by *(1)* in Figure 3). Issue queue usage counters are decremented when instructions are issued for execution (2). Register usage counters are decremented when the instruction commits (3), hence the file register is left unchanged. Pending cache miss counters are incremented when new misses are detected (4), and decremented when misses are serviced (5). The activity counter is initialized to 256.[2] This counter is decremented each cycle if the thread does not allocate new entries of that type of resource, and reset to 256 if the thread requires that resource (6). If the counter reaches zero, we classify the thread as inactive for that resource.

Second, concerning the sharing model, DCRA also needs simple control logic to implement this. This logic provides fixed, pre-computed calculations and hence it does not need write logic. Each cycle the sharing model checks that the number of allocated entries of slow active threads does not exceed the number that has been assigned to them. If such a thread allocates more resources, it is fetch-stalled

[2] We use several values for this parameter ranging from 64 to 8192 and this value gives the best overall results.

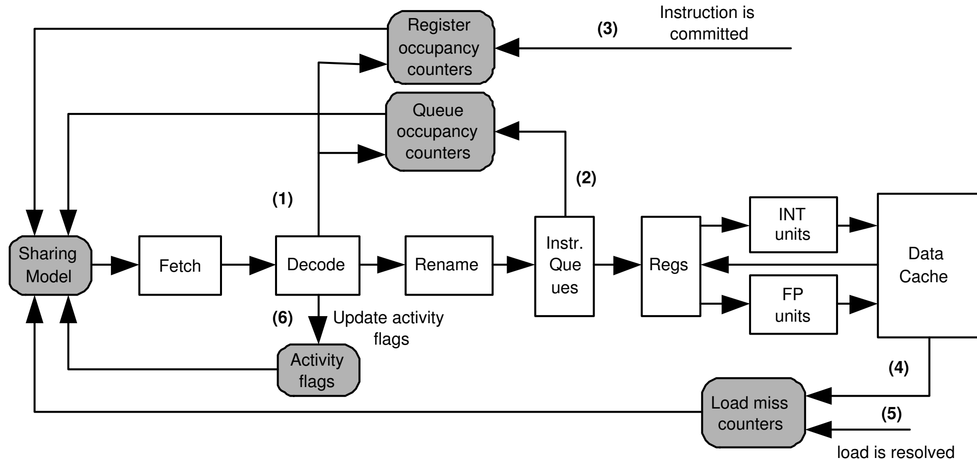


Figure 3. Possible implementation of our dynamic allocation policy

until it releases some of the allocated resources. Otherwise, it is allowed to enter the fetch stage and compete for resources. Recall that the fast-active threads are left unrestricted, being allowed to allocate as many resources as the S_A threads leave them, and the inactive threads are not allocating any entry for that resource.

The sharing model may be implemented in two ways:

- Using a combinational circuit implementing Formula 3 (the final resource allocation equation). The circuit receives as inputs the number of threads in each active group (F_A, S_A), 6 bits in case of a 4-context SMT. It provides the number of entries that each S_A threads is entitled to allocate.
- Alternatively, the sharing model could also be implemented with a direct-mapped, read-only table indexed with the number of S_A and F_A threads. For a 4-context processor, this table would have 10 entries. Changing the sharing model would be as easy as loading new values in this table. This is convenient, for example, when the memory latency changes.

Notice that we need two different circuits: one for the IQs and one for the registers.

4 Methodology

To evaluate the performance of the different policies, we use a trace driven SMT simulator derived from SMTSIM [16]. The simulator consists of our own trace driven front-end and an improved version of SMTSIM's back-end. The simulator allows executing wrong path instructions by using a separate basic block dictionary that contains all static instructions. Table 2 shows the main parameters of the simulated processor. This processor configuration represents a standard and fair configuration according to state-of-the-art papers in SMT.

We have fixed the number of physical register instead of the number of rename register. We use a register file of 320 physical registers, which means that we have $160 = 320 - (32 \times 4)$ rename registers when 4 threads are run, 224 when there are 3 threads, and 256 when there are 2 threads. On the other hand, in order to take into account timing effects of the register file, we assume two-cycle accesses.

Processor Configuration	
Pipeline depth	12 stages
Fetch/Issue/Commit Width	8
Queues Entries	80 int, 80 fp, 80 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	352
(shared)ROB size	512 entries
Branch Prediction Configuration	
Branch Predictor	16K entries gshare
Branch Target Buffer	256-entry, 4-way associative
RAS	256 entries
Memory Configuration	
Icache, Dcache	64 Kbytes, 2-way, 8-bank, 64-byte lines, 1 cycle access
L2 cache	512 Kbytes, 8-way, 8-bank, 64-byte lines, 20 cycle access
Main memory latency	300 cycles
TLB miss penalty	160 cycles

Table 2. Baseline configuration

Traces of the benchmarks are collected of the most representative 300 million instruction segment, following an idea presented in [13]. We use all programs from the SPEC2000 integer and fp benchmark suite. Each program is executed using the reference input set and compiled with the $-O2 - non_shared$ options using DEC Alpha AXP-21264 C/C++ compiler. Programs are divided into two groups based on their cache behavior (see Table 3): those with an

# of threads	Thread type	Workload group 1	Workload group 2	Workload group 3	Workload group 4
2	ILP	gzip, bzip2	wupwise, gcc	fma3d, mesa	apsi, gcc
	MIX	gzip, twolf	wupwise, twolf	lucas, crafty	equake, bzip2
	MEM	mcf, twolf	art, vpr	art, twolf	swim, mcf
3	ILP	gcc, eon, gap	gcc, apsi, gzip	crafty, perl, wupwise	mesa, vortex, fma3d
	MIX	twolf, eon, vortex	lucas, gap, apsi	equake, perl, gcc	mcf, apsi, fma3d
	MEM	mcf, twolf, vpr	swim, twolf, equake	art, twolf, lucas	equake, vpr, swim
4	ILP	gzip, bzip2, eon, gcc	mesa, gzip, fma3d, bzip2	crafty, fma3d, apsi, vortex	apsi, gap, wupwise, perl
	MIX	gzip, twolf, bzip2, mcf	mcf, mesa, lucas, gzip	art, gap, twolf, crafty	swim, fma3d, vpr, bzip2
	MEM	mcf, twolf, vpr, parser	art, twolf, equake, mcf	equake, parser, mcf, lucas	art, mcf, vpr, swim

Table 4. Workload classification based on cache behavior of threads.

Benchmark type	Benchmark name	L2 cache miss rate
INTEGER	mcf	29.6
	twolf	2.9
	vpr	1.9
	parser	1.0
FP	art	18.6
	swim	11.4
	lucas	7.47
	equake	4.72

(a) MEM threads

Benchmark type	Benchmark name	L2 cache miss rate
INTEGER	gap	0.7
	vortex	0.3
	gcc	0.3
	perl	0.1
	bzip2	0.1
	crafty	0.1
	gzip	0.1
	eon	0.0
FP	apsi	0.9
	wupwise	0.9
	mesa	0.1
	fma3d	0.0

(b) ILP threads

Table 3. Cache behavior of each benchmark

L2 cache miss rate higher than 1% are considered memory bounded (MEM). The others are considered ILP. It is vital to differentiate among program types and program phases. The program type concerns the L2 miss rate. Obviously, a MEM program experiences many slow phases, more than an ILP program. However, ILP programs also experience slow phases and MEM programs fast phases.

The properties of a workload depend on the number of threads in that workload and the memory behavior of those threads. In order to make a fair comparison of our policy, we distinguish three types of workloads: ILP, MEM, and MIX. ILP workloads contain only high ILP threads, MEM workloads contain only memory-bounded threads (threads with a high L2 miss rate), and MIX workloads contain a mixture of both. We consider workloads with 2, 3, and 4 threads. We do not include workloads with more than 4 threads because several studies [5, 6, 16] have shown that for workloads with more than 4 contexts, performance saturates or even degrades. This situation is counter productive because cache and branch predictor conflicts counteract the additional ILP provided by the additional threads.

A complete study of all benchmarks is not feasible due to excessive simulation time: all possible combinations of 2, 3 and 4 benchmarks give more than 10,000 workloads. We have used the workloads shown in Table 4. Each workload is identified by 2 parameters: the number of threads it contains and the type of these threads (ILP, MIX, or MEM). Hence, we have 9 workload types. As can be seen in Table 4, we have built 4 different groups for each workload type in order to avoid that our results are biased toward a specific set of threads. Benchmarks in each group have been selected randomly. In the result section, we show the average results of the four groups, e.g., the MEM2 result is the mean of the `mcf+twolf`, `art+vpr`, `art+twolf`, and `swim+mcf` workloads.

5 Performance evaluation

We compare our DCRA policy with some of the best fetch policies currently published: ICOUNT [15], STALL[14], FLUSH[14], FLUSH++[1], DG[2] and PDG[2]. Our results show that for the setups examined in this paper, FLUSH++ outperforms both STALL and FLUSH, and DG outperforms PDG. Hence, for brevity, we only show the results for ICOUNT, FLUSH++, and DG. We also compare DCRA with a static resource allocation that evenly distributes resources among threads.

Several performance metrics have been proposed for SMT processors. Some of these metrics try to balance throughput and fairness [10]. We use separate metrics for the raw execution performance and for execution fairness. For performance, we measure IPC throughput, the sum of the IPC values of all running threads, as it measures how effectively resources are being used. However, increasing IPC throughput is only a matter of assigning more resources to the faster threads and hence measuring fairness becomes imperative. We measure fairness using the $Hmean$ metric proposed in [10], as it has been shown that it offers better fairness-throughput balance than $Weighted\ Speedup$ [14]. Hmean measures the harmonic mean of the IPC speedup (or slowdown) of each separate thread, exposing artificial

throughput improvements achieved by providing resources to the faster threads.

5.1 Dynamic vs. static allocation

In this section, we compare DCRA with a static model in which each thread is entitled to use an equal share of resources. A recent study [12] quantifies the impact of partitioning the IQs and other shared resources in an SMT processor. Regarding the IQs, the authors reach two important conclusions. First, moving from a fully shared IQ to a evenly divided IQ has a negligible impact on performance. Second, they conclude that it is quite challenging to obtain significant benefits from a non-uniform IQ allocation.

We agree that a non-uniform allocation of the IQs does not provide significant benefits, but only if this is done without considering dynamic program behavior. That is, if this is done in a fixed way for the entire execution of the program. However, our dynamic sharing model provides a non-uniform issue queue allocation, where resource allocation varies with program phases (programs with temporarily more resource requirements are entitled to use some resources of threads with lower requirements) and where threads not using a resource give their share to the other threads.

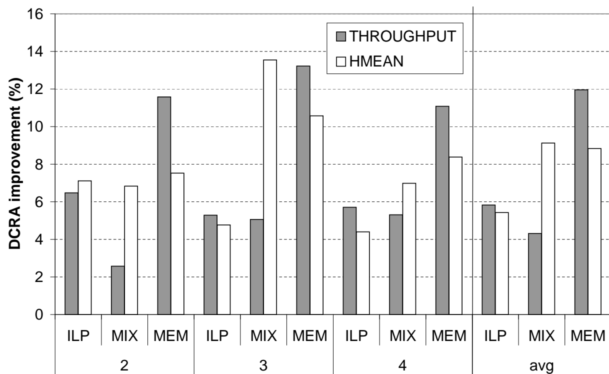


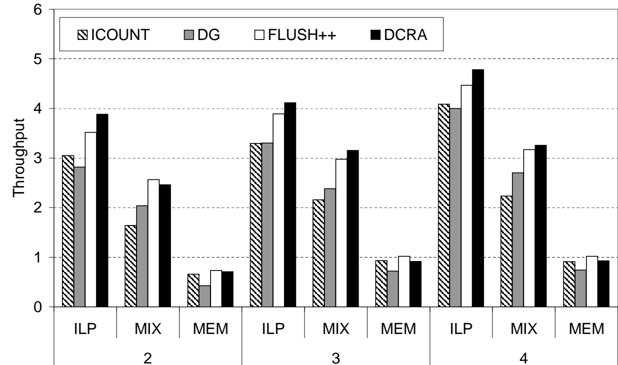
Figure 4. Throughput/Hmean results of DCRA compared to static resource allocation

Figure 4 shows the improvement of our dynamic model over the static one. We observe that our dynamic model outperforms the static model for all workloads: 7% in throughput and 8% in fairness, on average.

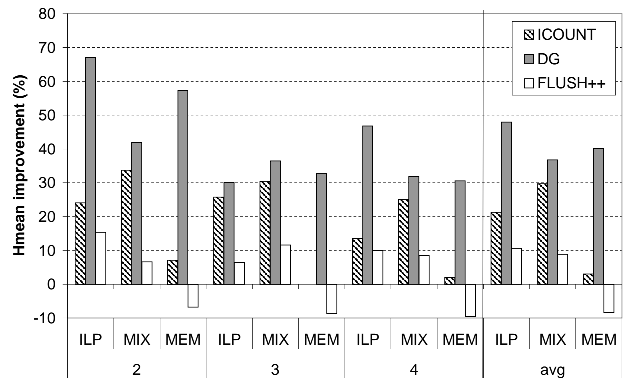
We also observe that the improvements of the dynamic over the static model are higher for the MIX workloads. In order to provide more insight in this issue, Table 5 shows how often threads in 2-thread workloads are either in the same phase or in different phases. The key point is that DCRA is more effective than SRA when threads are in different phases, the most common case for the MIX workloads (63% of the time). For ILP and MEM workloads, this situation is not so common, see Table 5. However, DCRA also is efficient for ILP and MEM workloads because it also classifies threads according to resource usage.

WORKLOAD TYPE	SLOW - SLOW	FAST-SLOW SLOW-FAST	FAST - FAST
ILP	7.8	41.4	50.8
MIX	25.6	63.2	11.2
MEM	85.0	14.7	0.3

Table 5. Distribution of threads in phases for 2-thread workloads



(a) IPC throughput



(b) Hmean fairness improvement

Figure 5. Throughput/Hmean improvement of DCRA over ICOUNT, FLUSH++, and DG.

5.2 DCRA vs. I-fetch policies

In this subsection, we compare the DCRA policy with ICOUNT, FLUSH++, and DG.

Figure 5(a) shows the IPC throughput achieved by DCRA and the other fetch policies. We observe that DCRA achieves higher throughput than any of the other fetch policies for all workloads, except for FLUSH++ in the MEM

workloads. The advantage of FLUSH++ over DCRA is due to the fact that for the MEM workloads, especially for the 4-MEM workloads, there is an overpressure on resources: there is almost no throughput increase when going from the 3-MEM workloads to the 4-MEM workloads. As a consequence of this high pressure on resources, it is preferable to free resources after a missing load than try to help a thread experiencing cache misses. On average, DCRA improves ICOUNT by 24%, DG by 30%, and FLUSH++ by 1%.

Regarding Hmean results, shown in Figure 5(b), DCRA improves all other policies. On average, DCRA improves FLUSH++ by 4%, ICOUNT by 18% and DG by 41%. Again, the FLUSH++ policy performs better than DCRA in the MEM workloads, for the same reasons described above.

However, the slight performance advantage of FLUSH++ over DCRA in the MEM workloads comes at a high cost: every time a thread is flushed to reclaim its resources for the other threads, instructions from the offending thread must be fetched, decoded, renamed, and even sometimes executed again. We have measured this overhead, and for 300 cycles of memory latency, FLUSH++ fetches 108% more instructions than DCRA. That is a $2X$ increase in activity for the processor's front-end.

The advantage of DCRA over the other resource-conscious fetch policies is that it allows the memory-bound thread to continue executing instructions with an increased -but limited- resource share. This increased resource assignment allows the thread to launch more load operations before stalling due to resource abuse, and increases the memory parallelism of memory-bound applications while high ILP ones do not suffer much (as shown in Figure 2). We have measured the increase in the number of overlapping L2 misses while using DCRA compared to using FLUSH++, and we have found an average increase of 18% in the memory parallelism of the workloads (22% increase in ILP workloads, 32% in MIX workloads, and 0.5% in MEM workloads).

Further analysis of the MEM workloads shows that DCRA is adversely affected by degenerate cases like *mcf*. Our results show a 31% increase in the number of overlapping misses for *mcf*, however, this increase is hardly visible in the overall processor performance due to the extremely low baseline performance, and comes at the expense of slightly decreased performance of other threads. That explains why FLUSH++ handles *mcf* better than DCRA, giving it the advantage in MEM workloads. Future work will try to detect these degenerate cases in which assigning more resources to a thread does not contribute at all to increased overall results or results in overall performance degradation.

5.3 Sensitivity to resources

In this section, we show how the improvement of DCRA over other alternatives depends on the amount in resources of the processor. It seems obvious that if we increase the amount of resources, sharing them among threads should be an easier task, as we diminish the risk that threads starve for lack of resources. However, we show that long latency events (such as L2 cache misses) can still cause resource monopolization by a single thread, regardless of the amount of resources available.

Register file

Figure 6 shows the average performance improvement of DCRA over ICOUNT, FLUSH++, DG, and SRA, as we change the number of physical registers from 320 to 384 entries. For this experiment, we have used 80-entry queues and a memory latency of 300 cycles.

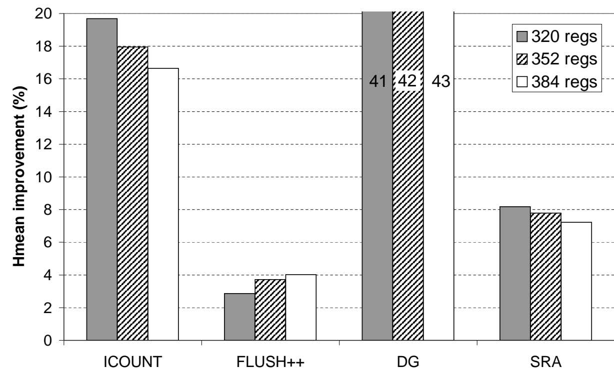


Figure 6. Hmean improvement of DCRA over other mechanisms as we change the register pool size.

We see that as we increase resources, the performance advantage of DCRA over SRA and ICOUNT diminishes since each thread receives more resources and the possibilities for starvation are reduced.

Regarding DG, we observe that as we increase the amount of resources, the advantage of DCRA also increases. This is caused by the fact that as we increase the size of the register pool, stalling threads on every L1 miss leads to a higher resource under-use. The comparison with FLUSH++ indicates a similar result: the objective of FLUSH++ is to make resources available to other threads after a missing load. While these deallocated resources may be necessary when there are few register, they become less important when the amount of resources is increased. We conclude that as we increase the register file size, the

amount of resource under-use introduced by FLUSH++ also increases, making DCRA a better option.

We also performed a similar analysis varying the size of the Instruction Queues, and obtained very similar results and conclusions. These are not shown due to lack of space.

Memory latency

As we have seen in this paper, memory-bounded threads require many resources to exploit ILP and will not release them for a long time. We now examine how the memory latency has an impact on the performance of DCRA and the other policies considered.

Figure 7 shows the average performance improvement of DCRA over the other policies as we change the change the memory latency from 100 to 300, and 500 cycles and the L2 latency from 10 to 20, and 25 cycles. For this experiment, we use 352 physical registers and 80-entry queues. Note that as latency increases we must be less aggressive sharing resources to SLOW threads as they retain these resources for longer time. In order to take into account this fact we use a different sharing factor, C, for each latency. For the 100-cycle latency the best results are obtained when $C = 1/T$. For a latency of 300 cycles when C=1/(T+4). Finally for the 500-cycle latency we use a sharing factor $C = 0$ for the IQs and $C = 1/(T + 4)$ for the registers.

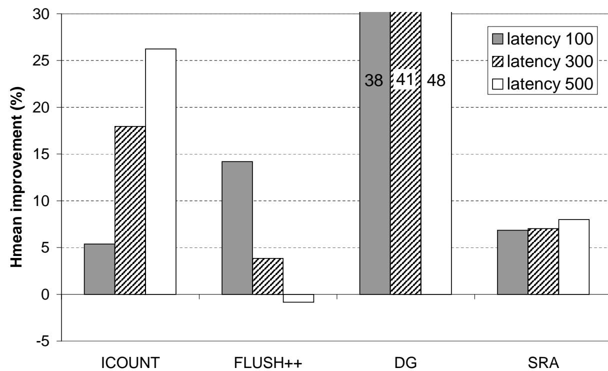


Figure 7. Hmean improvement of DCRA over fetch policies and SRA as we change the memory latency.

We observe that both DCRA and SRA suffer a similar penalty as the memory latency increases, but that both policies are still safe against the problem of resource monopolization. However, DCRA slightly increases its performance advantage due to its ability to dynamically move resources from threads which can do without them to threads which really can put them to use.

The results for ICOUNT show that it suffers a high performance penalty as the memory latency increases, as it is the only policy that does not take into account the memory behavior of threads. With increasing memory latencies, the problem of resource monopolization becomes even more severe, as the resources will not be available to others for even longer periods of time.

DCRA also improves its performance compared to DG when the memory latency increases. FLUSH++ is the only policy which reduces the performance advantage of DCRA as the latency increases. Given that FLUSH++ actually deallocates the resources of a thread missing in L2 and makes them available again to the remaining threads, it is able to use resources more effectively as they are allocated on-demand. The threads which did not miss in cache can use all the processor resources to exploit ILP. As much as DCRA prevents resource monopolization, the resources allocated by a missing thread are still not available to the other threads.

However, as we mentioned before, this increased flexibility in resource allocation comes at the cost of significant increases in the front-end activity. Instructions from the flushed thread have to be fetched, decoded, renamed, and in some cases re-executed after the missing load is resolved. Our measurements indicate a 108% increase in front-end activity for 300 cycles of memory latency, and a 118% increase for 500 cycles. If we account for the $2X$ increase in front-end activity and the negative effect of degenerate cases like *mcf* on DCRA performance (which we expect to fix in future work), we believe that DCRA offers a better alternative than FLUSH++.

From these results, we conclude that DCRA offers improved throughput and fairness balance for moderately sized processors. Moreover, as we increase the amount of available resources and the memory latency, which is currently happening in high performance processors, the importance of correctly managing resources increases, making DCRA an even better alternative for future SMT designs.

6 Conclusions

The design target of an SMT processor determines how shared resources should be shared. If a fair treatment of all threads is required, then a static partitioning of resources is an attractive design choice. If IPC throughput is to be valued above all else, a dynamic partitioning of resources where all threads compete for a pool of shared resources is required. Current dynamically partitioned designs depend on the fetch policy for resource allocation. However, the fetch policy does not directly control how many resources are allocated to a thread and current policies can cause both resource monopolization and resource under-use, obtaining less than optimal performance.

We have proposed to use a direct resource allocation policy, instead of fully relying on the fetch policy to determine

how critical resources are shared between threads. Our dynamic resource allocation technique is based on a dynamic classification of threads. We identify which threads are competing for a given resource and which threads should be able to give part of their resources to other threads without damaging performance. Our technique continuously distributes resources taking these classifications into account and directly ensures that no resource-hungry thread exceeds its rightful allocation.

Our results show that DCRA outperforms both static resource allocation and previously proposed fetch policies for all evaluated workloads. Throughput results show that DCRA improves SRA by 8%, ICOUNT by 24%, DG by 30%, and FLUSH++ by 1%, on average. The average Hmean improvement of DCRA is 7% over SRA, 18% over ICOUNT, 41% over DG, and 4% over FLUSH++. These results confirm that DCRA does not obtain the ILP boost by unfairly preferring high ILP threads over slower memory-bounded threads. On the contrary, it presents a better throughput-fairness balance. Summarizing, we propose a dynamic resource allocation policy that obtains a better throughput-fairness balance than previously proposed policies, making it an ideal design point for both throughput and fairness oriented SMT designs.

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contract TIC-2001-0995-C02-01, and grant FP-2001-2653 (Francisco J. Cazorla), the HiPEAC European Network of Excellence, and an Intel fellowship. The authors would like to thank Peter Knijnenburg for his comments and Oliverio J. Santana, Ayose Falcón, and Fernando Latorre for their work in the simulation tool. The authors also would like to thank the reviewers for their valuable comments and Brad Calder for his help in the camera ready of this paper.

References

[1] F. J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Improving memory latency aware fetch policies for SMT processors. *Proceedings of the 5th International Symposium on High Performance Computing*, Oct. 2003.

[2] A. El-Moursy and D. Albonesi. Front-end policies for improved issue efficiency in SMT processors. *Proceedings of the 9th International Conference on High Performance Computer Architecture*, Feb. 2003.

[3] R. Espasa and M. Valero. Multithreaded vector architectures. *Proceedings of the 3rd International Conference on High Performance Computer Architecture*, pages 237–249, Feb 1997.

[4] R. Goncalves, E. Ayguade, M. Valero, and P. O. A. Navaux. Performance evaluation of decoding and dispatching stages in simultaneous multithreaded architectures. *In Proceedings of the 13th Symposium on Computer Architecture and High Performance Computing*, Sep 2001.

[5] M. Gulati and N. Bagherzadeh. Performance study of a multithreaded superscalar microprocessor. *Proceedings of the 2nd International Conference on High Performance Computer Architecture*, pages 291–301, Feb. 1996.

[6] S. Hily and A. Seznec. Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading. Technical Report 1086, IRISA, Feb. 1997.

[7] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 136–145, May 1992.

[8] R. Kalla, B. Sinharoy, and J. Tendler. SMT implementation in POWER 5. *Hot Chips*, 15, Aug 2003.

[9] C. Limousin, J. Sebot, A. Vartanian, and N. Drach-Temam. Improving 3D geometry transformations on a simultaneous multithreaded SIMD processor. *Proceedings of the 15th International Conference on Supercomputing*, May 2001.

[10] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, Nov. 2001.

[11] D. T. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), Feb 2002.

[12] S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on SMT processors. *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 15–25, Sept. 2003.

[13] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[14] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreaded processor. *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, Dec. 2001.

[15] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. *Proceedings of the 23th Annual International Symposium on Computer Architecture*, pages 191–202, Apr. 1996.

[16] D. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *Proceedings of the 22th Annual International Symposium on Computer Architecture*, 1995.

[17] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. *Proceedings of the 1st International Conference on High Performance Computer Architecture*, pages 49–58, June 1995.

[18] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.