

# Approaching a Smart Sharing of Resources in SMT Processors

Francisco J. Cazorla<sup>1</sup>, Enrique Fernandez<sup>2</sup>, Alex Ramirez<sup>1</sup>, Mateo Valero<sup>1</sup>

<sup>1</sup>DAC, UPC, Spain, {fcazorla,aramirez,mateo}@ac.upc.es

<sup>2</sup>University of Las Palmas de Gran Canaria, Spain, efernandez@dis.ulpgc.es

## Abstract

SMT processors increase performance by executing instructions from several threads simultaneously. These threads use the processor's resources 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 throughput and also the performance of each individual thread.

Currently, the processor instruction fetch policy decides each cycle which threads enter the processor to compete for resources. However, these fetch policies **only** use indirect indicators of how resource allocation is carried out. This may cause resource monopolization by a single thread, or wasted resources when no thread can use them. Both situations can harm the processor performance and occur, for example, after an L2 cache miss.

This paper is a first step toward dynamic resource allocation for SMT processors. We show that being conscious about resource demand and directly controlling resource assignment significantly improves performance of SMTs. We introduce for the first time the concept of *resource allocation policy* in order to provide such a control. Our results show that our resource allocation policy outperforms the best published fetch policies for throughput and fairness, like FLUSH, by 7% on average. In addition, our resource allocation policy does not need to squash instructions from the pipeline, like FLUSH, in order to get this performance improvement. As a result, it reduces dynamic power consumption and hardware complexity.

## 1 Introduction

Superscalar processors increase performance by exploiting instruction level parallelism (ILP) within a single application. However, data and control dependences reduce the available ILP in 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 processors (SMT) 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 [4][11][12][13] with a moderate area overhead over a superscalar processor [1][2][5][8]. However, we should not forget that 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 at almost 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 fastest 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 be severely impaired. Therefore, to ensure that all threads are treated fairly is also a desirable objective for an SMT processor that cannot be quickly disregarded.

In current SMT processors, threads are executed in a common resource pool and are allowed to freely compete for resources. The instruction fetch (I-fetch) policy determines how this competition is carried out: each clock cycle, it chooses which threads can enter the processor and are the first ones to get the opportunity to use available resources. However, current fetch policies do not exercise direct control over how resources are distributed among threads. They use only indirect indicators of potential resource abuse by a given thread, for example, after L2 cache misses. Because no direct control over resources is exercised, it is still possible that a thread allocates 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 allocated most of the resources will not release them for a long period of time. There are fetch policies in the literature [3][6][10] 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, with varying degrees of success. The main problem of these policies is that in their attempt to prevent resource monopolization they may introduce resource under-use, because they are preventing a thread from using a set of 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 on resource allocation is exercised. At any given time, threads must be forced to use a limited amount of resources. Otherwise, they could monopolize shared resources. In order to control the amount of resources given to each thread, we introduce the concept of *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 very different resource demands. Even more, a given program has different resource demands during 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. Our results show that our resource allocation policy improves the best published I-fetch policies, like FLUSH, in throughput as well as in fairness [7]. This improvement is 7% on average. In addition, the resource allocation policy presented in this paper does not

require to squash instructions in the pipeline, and hence, reduces hardware complexity, energy and power consumption.

The remainder of this paper is structured as follows. We present related work in Section 2. In Section 3, we present our policy. Section 4 is devoted to comparing the resource allocation policies and instruction fetch policies. In section 5, we explain our experimental environment. Section 6 presents our simulation results. Conclusions are given in Section 7. Finally, Future Work is presented in section 8.

## 2 Related work

Many I-fetch policies in SMT processors have been proposed. Most of these policies use L1/L2 data misses as indicators of a possible resource monopolization.

ROUND-ROBIN [11] is the most basic fetch policy and simply fetches instructions from all threads alternatively, disregarding the resource use of each thread. This policy does not control any resource distribution.

ICOUNT [11] prioritizes threads with fewer instructions in the pre-issue stages and presents good results for threads with high ILP. However, an SMT has difficulties with threads that experience many loads that miss in the L2 cache. 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 [10] is built on top of ICOUNT to avoid the problems caused by threads with a high cache miss rate. It detects that a thread has a pending L2 miss and prevents the thread from fetching further instructions to avoid resource abuse. However, L2 cache misses are only an indicator of possible resource abuse. On the one hand, L2 miss detection may be already too late to prevent a thread from allocating most of the available resources. On the other hand, it is possible that the resources allocated to a thread are not required by any other thread, and hence the thread could very well continue fetching instead of stalling, producing resource under-use.

FLUSH [10] 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 blocked by 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. In [10] the authors show that FLUSH outperforms STALL.

Data Gating [3] is a recently proposed policy that attempts to reduce the effects of loads missing in the L1 data cache by stalling threads on each L1 data miss. However, not all L1 misses cause an L2 miss. Our results (not shown here) indicate 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 [3] works like STALL, that is, it prevents a thread from fetching instructions as soon as a cache miss is predicted. By using a miss predictor, it avoids detecting the cache misses too

late, but it introduces 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 very hard to predict accurately [14], reducing the advantage of this technique.

### 3 Approaching a resource allocation policy

A key point for a resource allocation policy is that every program has different resource demands. Moreover, most programs go through different behavior patterns during their execution. In each pattern, resource demands of a program may vary drastically. In order to provide better results, a resource allocation policy must take into account these phases and thus the different resource demands of threads.

Figure 1 shows a diagram of how a resource allocation policy could work. The first step is to *classify* threads into groups. In this classification, threads in the same phase, and thus with similar resource requirements, are placed in the same group. The second step is to *allocate* resources to threads based on resource availability and the classification previously made. In the following two subsections we describe these phases in more detail.

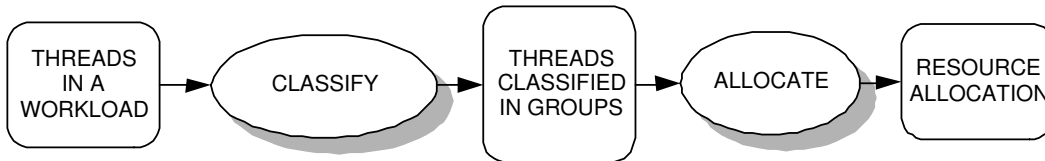


Figure 1: Main tasks of a resource allocation policy

#### 3.1 Thread classification

The first task for a resource allocation policy is to classify threads so that threads in each group have similar resource requirements. The objective is to provide information to the resource allocation mechanism (section 3.2) on the demand of resources. It is important to note that this classification is not done for the entire lifetime of threads. Instead, it is dynamic and identifies the dynamic changes in resource requirements.

In order to carry out this classification we use *indicators*. An indicator is an event that provides information on the future use of resources that a thread will make. In this paper we use only cache behavior of threads as indicator. After having explored several possibilities, we classify threads using L1 data cache misses as an indicator of cache behavior. Our classification mechanism classifies threads in two groups, FAST or SLOW.

- **The SLOW group.** Threads with pending L1 data misses are classified in the SLOW group, because they may allocate many resources for a long period of time: 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, mostly 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.

- **The FAST group.** Threads with no pending L1 data cache misses are classified in the FAST group, because they are able to run using a small set of resources that are rapidly re-used. That is, FAST threads are able to exploit ILP with few resources. It is noted 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.

Please note the allocation mechanism uses this classification to make the final resource distribution among threads.

## 3.2 Resource allocation

The second task of a resource allocation policy is to allocate resources to threads. The main objectives of this allocation are two-fold.

- First, to avoid resource monopolization. This is achieved by enforcing hard limits in the use of shared resources. Any thread that uses more resources than assigned to it, is fetch stalled.
- Second, reduce resource under-use. This is achieved by giving more resources to threads in SLOW phases that have higher resource demands, as long as it does not affect threads in FAST phases.

Resources are allocated to threads depending on how many FAST and SLOW threads there are. In this paper, we only study the case in which there are two threads, and hence, there are 4 possible combinations.

- When both threads are in the same phase (FAST, FAST) or (SLOW, SLOW), our policy assumes that they have the same resource demands. Hence, it evenly divides resources to threads.
- When there is one thread in each group, (FAST, SLOW) and (SLOW, FAST), we know that the thread in the SLOW phase has more resource demands than the one in the FAST phase. In this situation, our policy gives more resources to the thread in the SLOW phase. For example, this thread could be given 60% of shared resources while the thread in the FAST phase only may use the remainder 40%.

In short, our policy starts by assigning an equal share of each shared resource when both threads have the same type. Next, when each thread is of a different type, the thread in an FAST phase shares part of its resources with the other thread. This way, the thread in a SLOW phase is assigned its equal share and also may borrow some additional resources from a thread which can run without them. We have experimented with different values for the amount of resources the thread in the SLOW phase can borrow from the FAST one. In our simulations we have varied this amount from 50% to 87.5% with an step of 12.5%, for both the IQs and the registers, as it is shown in Table 1.

Program Phase		Resource allocation(%)	
Thread 0	Thread 1	Thread 0	Thread 1
FAST	FAST	50	50
SLOW	SLOW	50	50
SLOW	FAST	50	50
		62.5	37.5
		75	25
FAST	SLOW	87.5	12.5
		50	50
		37.5	62.5
		25	75
		12.5	87.5

Table 1: Resource allocation used

## 4 Resource allocation policies vs. Instruction fetch policies

The main differences among a resource allocation (RAlloc) policy and an instruction fetch policy focus on two points: the response action and the input information involved.

- The input information is the information used by the policy to take 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 fetch stall a thread.

I-fetch policies just control the fetch bandwidth. All I-fetch policies we have seen 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. Table 2 shows the input information and response actions of the fetch policies presented in Section 2. In this table, ‘issue delay’ means that the policy detects that a load spends more time in the cache hierarchy than needed to access the L2 data cache.

An RAlloc policy control fetch bandwidth, as I-fetch policies do. As shown in [11], the fetch bandwidth is a key parameter for SMT processors. Hence the control of this resource is absolutely required. In addition, an RAlloc policy 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, RAlloc policies uses indirect indicators but, 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

I-Fetch policy	Input Information	Response Action
STALL	issue delay	stall fetch
FLUSH	issue delay	stall fetch and squash pipeline
DG	L1 data misses	stall fetch
PDG	L1 data misses and predicted misses	stall fetch

Table 2: Resource actions and input information of fetch policies

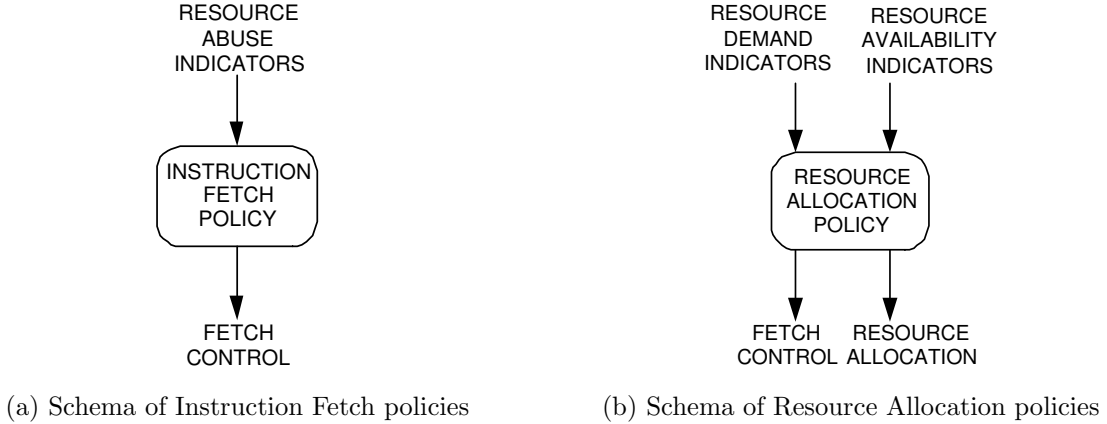


Figure 2: Schema of I-Fetch and RAlloc policies

abuse indicators and, as a consequence, when any of them is activated the I-fetch policy immediately stalls or flushes threads (see Figure 2 (a)). An RAlloc policy, see Figure 2 (b), perceives indirect indicators as information on resource demand. As a consequence, it does not immediately take measures on threads experiencing cache misses. Instead, it computes the overall demand as well as the availability of resources. Then, it splits shared resources and fetch bandwidth among 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.

In a more general approach, another important point to take into account is the number of threads running because this number determines the pressure on resources. The higher the number of threads, the higher the pressure. Current I-fetch policies disregard this information and as result the response action they take may be inadequate. RAlloc policies use this information when sharing resources between threads, and hence the resource allocation is according to demand of resources.

## 5 Methodology

To evaluate the performance of the different policies, we use a trace driven SMT simulator derived from SMTSIM [12]. 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 3 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. In our simulated processor both the IQs and the physical registers are shared among threads, while each thread has its own ROB.

Processor Configuration	
Pipeline depth	12 stages
Fetch/Issue/Commit Width	8
Queues Entries	32 int, 32 fp, 32 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	256
ROB size (each thread)	256
Branch Prediction Configuration	
Branch Predictor	16K entries gshare
Branch Target Buffer	256-enry, 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, 10 cycle access
Main memory latency	100 cycles
TLB miss penalty	160 cycles

Table 3: Baseline configuration

Traces are collected of the most representative 300 million instruction segment, following an idea presented in [9]. The workload consists of all programs from the SPEC2000 integer benchmark suite. Each program is executed using the reference input set and compiled with the  $-O2 - non\_shared$  options using the DEC Alpha AXP-21264 C/C++ compiler. Programs are divided into two groups based on their cache behavior (see Table 4): those with an L2 cache miss rate higher than 1%<sup>1</sup> are considered memory bounded (MEM). The rest 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

---

<sup>1</sup>The L2 and L1 miss rate are calculated with respect to the number of dynamic loads

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
	perlbmk	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 4: Cache behavior of isolated benchmarks

number	Workload type		
	ILP	MEM	MIX
1	gzip , bzip2	twolf , vpr	eon , twolf
2	eon , gap	mcf , parser	bzip2 , vpr
3	gcc , vortex	twolf , mcf	crafty , mcf
4	fma3d , apsi	lucas , equake	wupwise , art
5	wupwise , galgel	art , swim	lucas , galgel
6	crafty , mesa	twolf , swim	gap , art
7	perlbmk , mgrid	twolf , equake	wupwise , parser
8	bzip2 , apsi	mcf , art	gzip , swim

Table 5: Workload classification based on cache behavior of threads.

behavior of the individual threads. In order to make a fair comparison of our policy we distinguish three types of workloads: ILP, MEM and MIX. ILP workloads only contain high ILP threads, MEM workloads only contain memory-bounded threads (threads with a high L2 miss rate), and MIX workloads contain a mixture of both. In this paper, we consider workloads with only 2 threads like the Pentium 4 [8] or the Power 5 [5].

We have used the workloads shown in Table 5. We have built 8 different workloads for each group 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 next section we show the average results of the 8 workloads in each group.

## 6 Performance evaluation

We compare our smartRA policy with some of the best proposed fetch policies currently known: ICOUNT [11], STALL[10], FLUSH[10], DG[3] and PDG[3]. For clarity, we only show the results of those policies

that give better results for the setup examined in this paper: FLUSH, ICOUNT, and DG.

Several performance metrics have been proposed for SMT. Some of these metrics try to balance throughput and fairness [7]. We use separate metrics for the raw execution performance and for execution fairness. For performance, we measure IPC throughput, the sum of the 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 so measuring fairness becomes imperative. We measure fairness using the *Hmean* metric proposed in [7], as they show that it offers a better fairness-throughput balance than the *Weighted Speedup* [10]. Hmean measures the harmonic mean of the IPC speedup (or slowdown) of each separate thread, exposing artificial throughput improvements achieved by giving resources to the faster threads.

## 6.1 Exploring different resource allocations

In this section we explore the effect of the amount of shared resources given to threads in SLOW phases when there is one thread in each phase. We vary this percentage from 50% to 87.5% in steps of 12.5%, for both the IQs and the registers. That means that for the IQs we assign 16, 20, 24 and 28 entries, respectively, to the thread in a SLOW phase. Likewise, we assign 96, 120, 144 and 168 rename registers to such a thread.

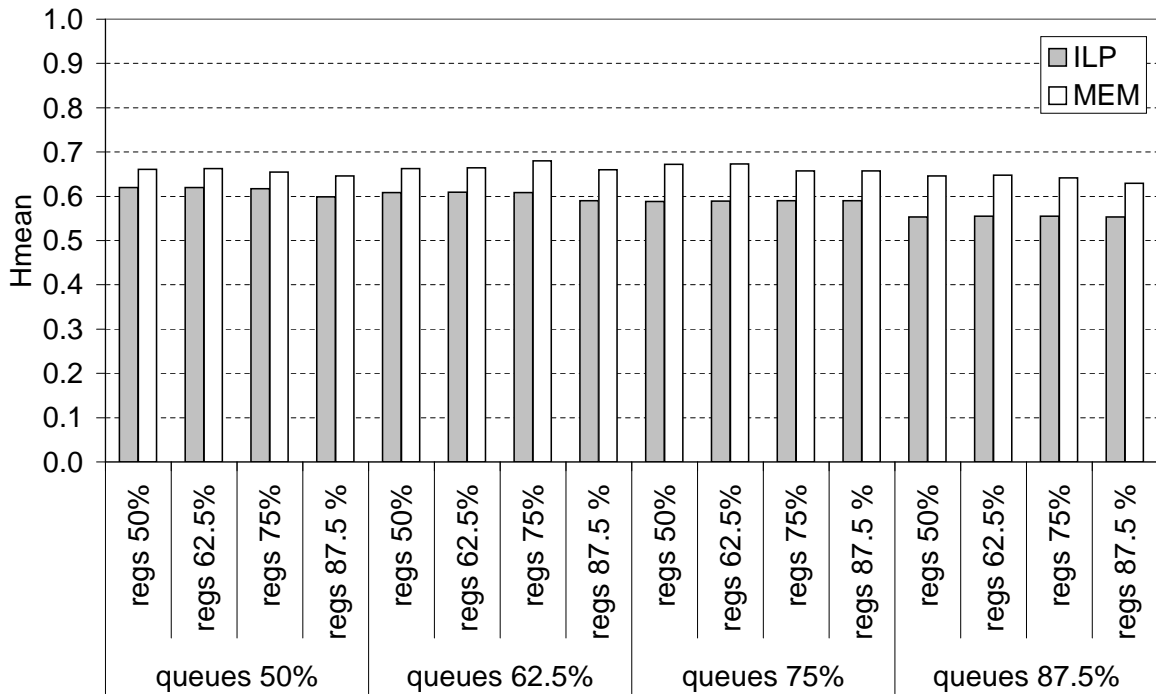


Figure 3: Comparison of the different resource allocations for ILP and MEM workloads

Figure 3 shows the Hmean obtained for different resource divisions for ILP and MEM workloads. As

expected the variation is low. The difference between the worst and the best IPC value is less than 16%. This is caused by the fact that for these workloads, most of time both threads are of the same type, either SLOW or FAST. As a result, most of the time resources are evenly divided over the threads.

In the case of MIX workloads, the situation where there is a thread in each group is much more frequent. Hence, the difference between the best and the worst value is higher. Figure 4 shows the Hmean results for each of the 8 MIX workloads for all resource allocations. We see that there is not a single resource allocation that leads to the best result for all workload types. Instead, each workload achieves the best Hmean result with a different resource allocation. Tables 6(a) and 6(b) give a deeper in-sight in this issue. These tables show for each workload type that IQ and register division that leads to the best Hmean result. For example, for the ILP1 workload the best Hmean result is achieved when the SLOW thread is allowed to use 50% of each issue queue and 75% of each register bank.

Workload type			
	ILP	MEM	MIX
1	50	50	50
2	50	50	50
3	50	50	50
4	50	87.5	62.5
5	50	87.5	75
6	50	62.5	62.5
7	50	62.5	50
8	87.5	87.5	50

(a) IQ entries

Workload			
	ILP	MEM	MIX
1	75	50	62.5
2	75	62.5	62.5
3	75	62.5	50
4	50	50	75
5	50	75	50
6	62.5	75	75
7	62.5	75	62.5
8	87.5	62.5	50

(b) Registers

Table 6: IQ and register division that lead to the best Hmean results. These divisions are applied when there is one thread in a SLOW phase and the other is in a FAST phase. Recall that if both threads are in the same phase resources are evenly split.

This experiment with different static resource allocations shows the need of a dynamic policy that for each particular workload selects a different resource allocation. We will assume that such a policy exists and would select in each case the best resource allocation. This is our policy that we compare with the I-fetch policies. We call our policy smartRA that stands for smart resource allocation.

In this initial study, we have used cache behavior as indicator. We really think that these results can be improved by using other indicators. Moreover, if we could make a different resource allocation for IQ type and register type, results can improve even more. We think that there is still much more room from improvement, which we will address in the future work section.

## 6.2 Resource allocation vs. I-fetch policies

Figures 5(a) and 5(b) show the throughput results obtained using smartRA and the I-fetch policies. We see that for ILP workloads the difference between the different policies is small, with smartRA always improving the other ones. This is mainly due to the fact that ILP threads rarely experience L2 misses. For

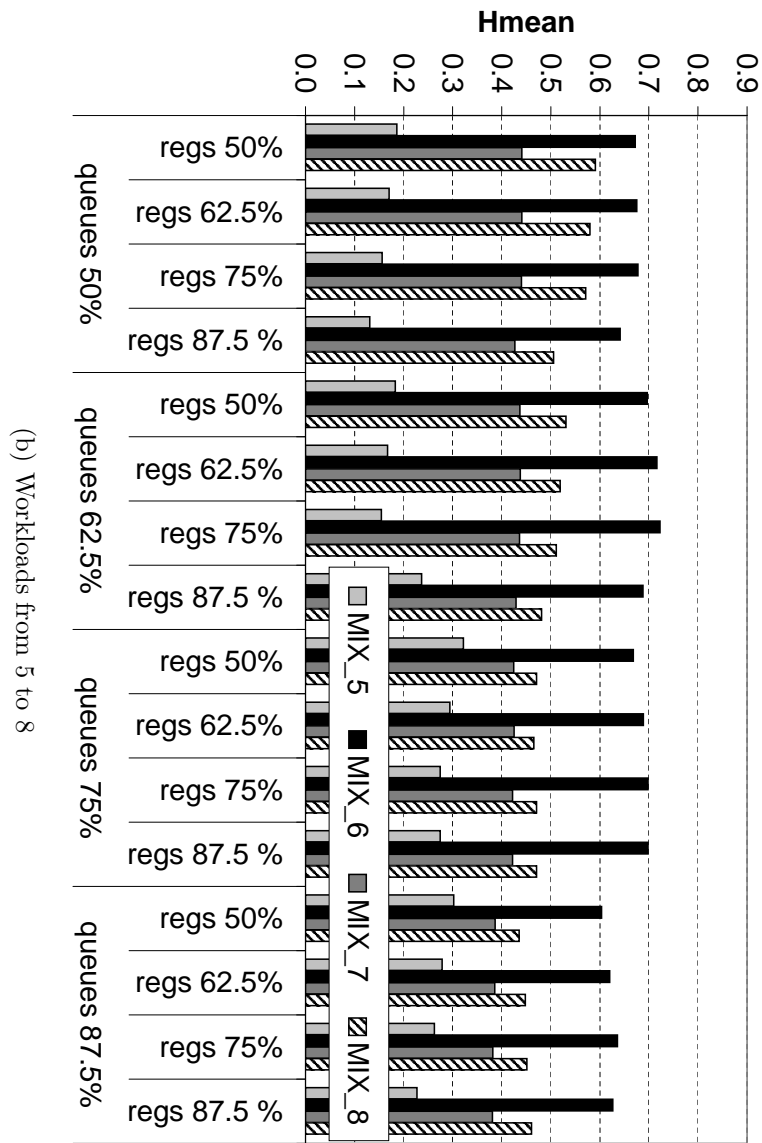
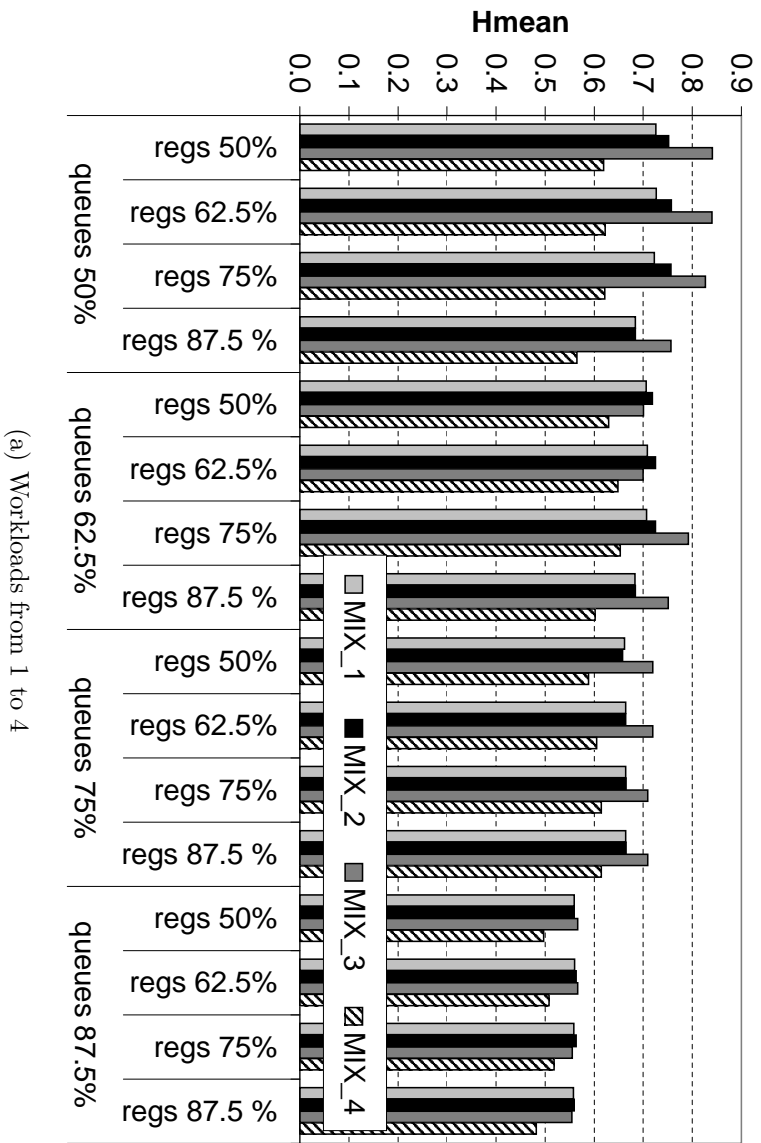
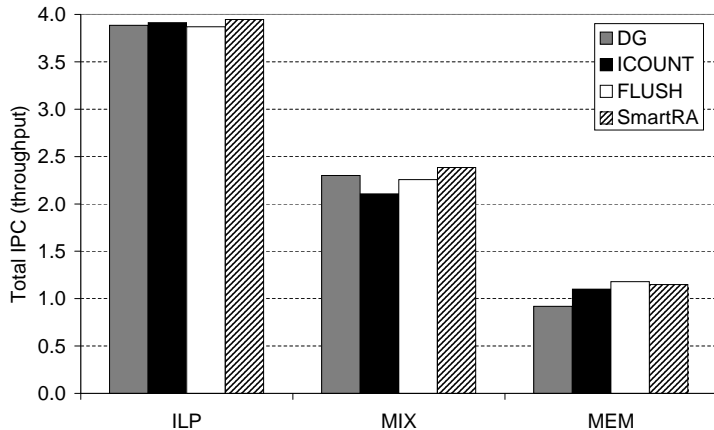
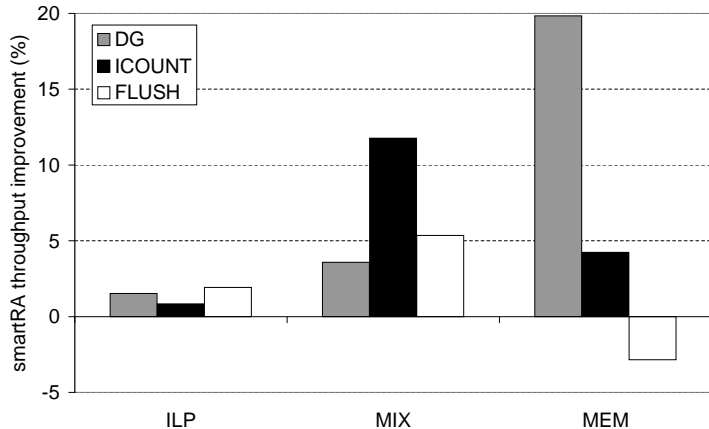


Figure 4: Hmean results for all resource allocations for all MIX workloads

the MIX and MEM workloads, smartRA improves all other policies except FLUSH for the MEM workloads where it suffers a slowdown of 3%. As we will see this is because these I-fetch policies favor threads in ILP phases over threads in MEM phases. On average, smartRA improves FLUSH by 1.6%, DG by 8.3%, and ICOUNT by 5.6% in throughput.



(a) Absolute values



(b) SmartRA improvements over various I-fetch policies

Figure 5: IPC throughput

Hmean results, see Figure 6, show that smartRA improves all other policies for all workload types. This indicates that smartRA is fairer than the other policies. This is caused by the fact that previously proposed policies favor ILP threads at the cost of degrading MEM threads. smartRA proceeds the other way around by helping threads in SLOW phases as they do not harm the performance of threads in ILP phases. As a result, smartRA improves FLUSH by 7.3%, DG by 13.5% and ICOUNT by 7.8%, on average in fairness.

From the fetch policies, FLUSH achieves the best results. On average, smartRA improves FLUSH by

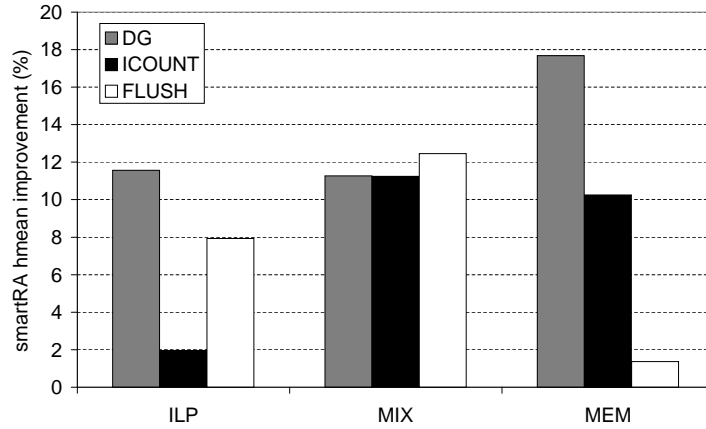


Figure 6: Hmean improvement of smartRA over various I-fetch policies

2% in throughput and 7% in fairness. In addition to this, smartRA has another advantage over FLUSH: it does not require squashing instructions in the pipeline. We have measured the amount of instructions that need to be re-fetched when the FLUSH policy is used, shown in Figure 7. In this figure, we are not taking into account the flushed instructions due to branch mispredictions, but only those related with loads missing in L2. We observe that for ILP workloads, this increase is the smallest one, since ILP workloads contain threads with small L2 miss rate. Nevertheless, the increase is significant, 12%. For the MIX workloads, it is almost 50%. It is worth mentioning that for MEM workloads the increase is about 87%. This means that the number of fetched instructions is almost duplicated, with the energy and power costs it implies. Notice that, when a squash of the pipeline is triggered, all squashed instructions have been fetched. In addition, some have been already decoded, some already mapped, some already queued, some already executed, etc., hence the wasted energy increases.

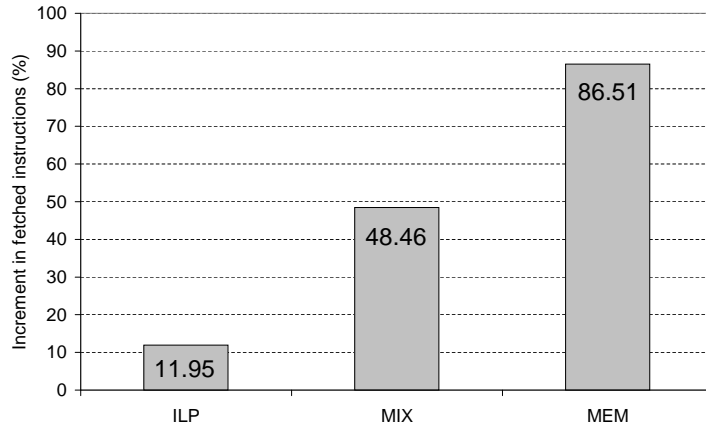


Figure 7: Increment in the number of fetched instructions when the FLUSH policy is used

## 7 Conclusions

In current SMT processors resources are assigned to threads as determined by the instruction fetch policy. However, the instruction fetch policy does not consciously control how many resources are allocated to threads. As a consequence, current policies cause both resource monopolization and resource under-utilization, wasting performance and energy.

This paper is a first step toward dynamic resource allocation for SMT processors. We have shown that in order to increase SMT performance direct control over shared resources, and moreover, adapting the resource allocation to program phases, and thus to program real resource demands, are required. The concept of *resource allocation policy* has been introduced for the first time in order to provide such control. Our results show that there is not an optimal static resource allocation for all workload types. On the contrary, each workload requires a different resource allocation, which makes evident the need of a dynamic policy, smartRA. After comparing smartRA with three of the best published fetch policies, we have shown that it outperforms all current designs, yielding processors with greater performance. Hmean results show that smartRA improves FLUSH by 7%, DG by 17% and ICOUNT by 11% on average for ILP, MIX, and MEM workloads. It is also remarkable that in order to get this performance improvement, smartRA does not need to squash instructions in the pipeline, like FLUSH. Hence, it reduces the re-fetched instruction effort (up to 87% in MEM workloads) and then, also reduces the dynamic power wasted and hardware complexity.

## 8 Future Work

This work is a first step toward a dynamic policy that provides a smart resource allocation based on real demand of threads. We think that there is still room for improvement. This improvement would come from the following points.

- The resource demands of threads depend on the phase that threads are in. In this work we have used cache behavior of threads, in particular data L1 misses, to identify the phase a thread is in and the amount of resources it needs. We think that there are other events that also indicate the resource demands threads. We are currently working on identifying these events.
- In this paper we have divided integer and floating point resources in the same way. That is, the integer issue queue and the floating point issue queue have been split equally. The same happens with the integer and fp physical registers. However, given that fp instructions have longer latencies than integer ones, fp codes have higher resource demands than integer ones. A different resource allocation for integer and fp resources could improve total performance.
- Furthermore, in the case of the issue queues, we think that the load/store queue requires a resource allocation different than the integer and fp one.

All these factors indicate to us that the results obtained in this paper can be improved and motivate us to research in these issues.

In addition to improve our resource allocation policy, in the future we plan to work in the following three issues. First, we will extend our idea for workloads with more than two threads. Second, we will propose a dynamic policy that fulfils the previous objectives. And third, we will propose a hardware implementation of our policy.

## 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, 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.

## References

- [1] J. Burns and J-L. Gaudiot. Quantifying the SMT layout overhead- does SMT pull its weight? *Proceedings of the 6th Intl. Conference on High Performance Computer Architecture*, pages 109–120, January 2000.
- [2] J. Burns and J-L. Gaudiot. SMT layout overhead and scalability. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):142–155, February 2002.
- [3] A. El-Moursy and D.H. Albonesi. Front-end policies for improved issue efficiency in SMT processors. *Proceedings of the 9th Intl. Conference on High Performance Computer Architecture*, February 2003.
- [4] 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 Intl. Symposium on Computer Architecture*, pages 136–145, May 1992.
- [5] R. Kalla, B. Sinharoy, and J. Tendler. SMT implementation in POWER 5. *Hot Chips*, 15, Aug 2003.
- [6] C. Limousin, J. Sebot, A. Vartanian, and N. Drach-Temam. Improving 3D geometry transformations on a simultaneous multithreaded SIMD processor. *Proceedings of the 15th Intl. Conference on Supercomputing*, pages 236–245, May 2001.
- [7] 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*, pages 164–171, November 2001.
- [8] D. T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), Feb 2002.
- [9] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. *Proceedings of the 10th Intl. Conference on Parallel Architectures and Compilation Techniques*, September 2001.

- [10] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreaded processor. *Proceedings of the 34th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, December 2001.
- [11] 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 Intl. Symposium on Computer Architecture*, pages 191–202, April 1996.
- [12] D.M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *Proceedings of the 22th Annual Intl. Symposium on Computer Architecture*, 1995.
- [13] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. *Proceedings of the 1st Intl. Conference on High Performance Computer Architecture*, pages 49–58, June 1995.
- [14] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. *Proceedings of the 26th Annual Intl. Symposium on Computer Architecture*, May 1999.