Quality of Service for Simultaneous Multithreading Processors

(QoS for SMT Processors)

Francisco Javier Cazorla Almeida

2005

A thesis submitted in fulfillment of
the requirements for the degree of
DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

Quality of Service for Simultaneous Multithreading Processors

(QoS for SMT Processors)

Francisco Javier Cazorla Almeida

2005

Advisors: **Mateo Valero Cortés**
Universitat Politècnica Catalunya
Álex Ramírez Bellido
Universitat Politècnica Catalunya
Enrique Fernández García
Universidad de Las Palmas de Gran Canaria

Collaborators: **Peter M.W. Knijnenburg**
Leiden University
Rizos Sakellariou
University of Manchester

A thesis submitted in fulfillment of
the requirements for the degree of
DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

A los miembros de mi familia: Antonio, Brígida, Antonio L., Rosi,
Marita, Carmelo, Alejandro y David.

Acknowledgments

Quería dar gracias en primer lugar a los dos principales culpables de que yo comenzara esta empresa de realizar una tesis doctoral, Mateo y Enrique. También dar las gracias a todas aquellas personas que participaron en la realización de esta tesis, Alex, Peter y Rizos.

En estos años de doctorado he convivido con dos personas de una grandísima calidad humana Javi y Germán. Echaré de menos esas largas noches de conversaciones sobre los temas mas diversos que a uno se le puedan venir a la cabeza. Otra persona de una enorme calidad humana con el que he pasado, y espero pasar, largas y muy amenas horas hablando sobre la vida es Jaume Abella. Jaume es ya un canario más y pocos como él recordarán los asaderos que hemos realizado ;-).

Durante el doctorado, uno tiene mucho tiempo para pensar sobre el propio doctorado, quizás demasiado. Yo, para distraerme solía pensar sobre campos no relacionados con la arquitectura de computadores. En ciertas ocasiones se me ocurrieron ideas sobre algunos de estos campos. Germán se encargo de mostrarme, para cada una de esas ideas, referencias que mostraban que ya habían sido publicadas. Estos desagredecimientos van dedicados a él :-). Por suerte, nunca le conté las ideas que tuve en mi campo de investigación.

También quería hacer una mención en esta sección a la cantidad enorme de personas que conocí en estos años. La mayoría los conocí en alguna de las ediciones del “Asadero dia de Canrias”. Intentare ahora abarcar al mayor numero de ellos aquí, perdón para todos aquellos que ahora no recuerdo: Javi, Germán, Javi Zalamea, Isidro, Rakesh, Alex Pajuelo, Jordi, Lemma, Jaume, Josep María, Juanjo, Montse, Xavi M., Beatriz, Oliver, Ayose, Alba, Daniel, Fermín, Lorenzo, Ramón, Pedro, Jordi Gonzalez, Marco Antonio, Stephan, Fernando, Carmelo, Tana, Ruth, Maribel, Saray, Patry, Dacil, Marco, Carla, Charly, Ester, Carlos, Cristina, Isa, Toni, Manolo, Mariam, Alexis, Ale, Fayna, Alex Muntada, Eva, Horacio, Susana, Adrian, Ruben Gonzalez, Raimir, Miquel, Carmina, Nayeli, Lizbeth, Ruben, Carme, Carmen, Vicenc, Eva, Marc, Anahí, Marsal ...

Finalmente, una parte que me gustaría se recordara de esta Tesis es la de “frases célebres”. Esta parte, que aquí comienza, es básicamente un listado de frases que marcaron, de una u otra forma, mi doctorado.

- Jaume Abella: (En una presentación) “Todo lo que he contado hasta ahora es mentira...”.

- Alex Pajuelo “Deberíais dar gracias a vuestros directores de Tesis por sacaros de la bebida de lunes a viernes”.
- Daniel Ortega: Por su teoría de la posibilidad en lugar de la teoría de la probabilidad.
- Carme Llopart: “La informática es parecida a las mujeres, todo es cuestión de encontrar los botones adecuados”.
- Germán Rodríguez: “Sepa señorita que estos son mis principios! Pero si no le gustan, tengo otros’.’
- Rubén González Benítez: “Si saben contar, que no cuenten conmigo”.
- Lemma Hundessa: “No acabes tu tesis hoy, de lo contrario no tendrás nada que hacer mañana”.

Abstract

Many processors take advantage of Instruction Level Parallelism (ILP) to execute several instructions from a single stream in parallel. However, data dependences and control flow instructions limit the available ILP. Several studies have shown that, in the near future superscalar processors may reach diminishing return points. Such studies also show that increasing the number of transistors or scaling up such processors only slightly improves overall performance. This problem has motivated the research into alternative techniques to increase the performance of processors. Multithreaded processors overcome this problem by allowing the execution of several tasks on the same processor.

Current trends in computer architecture show that many future processors will have some form of multithreading. This includes processors for high-performance systems [36][47], network processors [3][4], and embedded processors for real-time systems [10][41]. Each of these computing systems has different objectives (requirements). For example, in an embedded real-time system the target of the system could be to execute a given thread before a deadline while maximizing the overall system performance and reducing power consumption.

SMT processors have a good cost/performance/power consumption ratio, which makes them suitable for many types of computing systems mentioned above. However, as demonstrated in this thesis, current multithreaded processors are designed so that they intrinsically assume part of the responsibility of the system. The key problem is that in some cases the targets of the multithreaded processor could be different from the requirements of the systems running on it. This lack of flexibility to fulfill system requirements makes the use of SMTs in these systems difficult.

In this thesis, we analyze the feasibility of SMT for some of these computing systems. We show that current SMT processors are not adequate to achieve all these objectives successfully. We propose architectural changes in order to allow SMT processors to be used in a wider range of systems. In addition, a significant amount of effort has been spent on discussing future requirements that systems will put on SMT processors, proposing solutions to accomplish these requirements.

Contents

1 Introduction **1**
1.1 SMT processors . . . 5
1.2 Thesis contributions . . . 6
1.3 Thesis structure . . . 7

2 Experimental Framework **9**
2.1 Evaluation tool . . . 11
2.2 Benchmarks . . . 13

3 QoS for High-Performance Systems with Implicit Resource Allocation **15**
3.1 Introduction . . . 17
3.2 Related work . . . 18
3.3 Metrics in high-performance systems . . . 20
3.4 Flush++ . . . 21
3.4.1 Experimental setup . . . 22
3.4.2 Load miss predictors in an SMT environment . . . 22
3.4.3 Comparing the current policies . . . 25
3.4.4 Improved policies . . . 26
3.4.5 The flush++ policy . . . 30
3.4.6 Conclusions . . . 31
3.5 DWarn . . . 33
3.5.1 Simulation methodology . . . 33
3.5.2 The Dcache Warn policy (DWarn) . . . 35
3.5.3 Performance evaluation . . . 39
3.5.4 DWarn on different architectures . . . 43
3.5.5 Conclusions . . . 44
3.6 Chapter summary . . . 46

4 From Implicit Resource Allocation to Explicit Resource Allocation **47**
4.1 Introduction . . . 49
4.2 Increasing throughput and fairness . . . 50
4.2.1 Resource allocation policies vs. instruction fetch policies . . . 52
4.2.2 Methodology . . . 54
4.2.3 Performance evaluation . . . 56
4.2.4 Conclusions . . . 60
4.3 Feasibility of QoS in SMT through explicit resource allocation . . . 61
4.3.1 QoS: A novel collaboration between OS and SMT . . . 61

4.3.2 QoS by resource allocation 62
4.3.3 Conclusions 65
4.4 Chapter summary 65

5 QoS for High-Performance Systems with Explicit Resource Allocation 67
5.1 Dcra 69
5.1.1 The dcra policy 70
5.1.2 Methodology 76
5.1.3 Performance evaluation 78
5.1.4 Conclusions 83
5.2 Chapter summary 84

6 QoS for Real-Time Systems 85
6.1 Introduction 87
6.2 Related work 89
6.3 Predictable Performance 91
6.3.1 Problem statement 92
6.3.2 QoS problem definition 94
6.3.3 Solution of the QoS problem 95
6.3.4 Experimental setup 103
6.3.5 Results when there is one PPT 103
6.3.6 Results when there are two PPTs 108
6.3.7 Conclusions 116
6.4 Low-Variability Performance 116
6.4.1 Background on real-time scheduling 118
6.4.2 Experimental environment 119
6.4.3 Dynamic resource partitioning 121
6.4.4 Simulation results 129
6.4.5 Implementation 131
6.4.6 Conclusions 133
6.5 Chapter summary 133

7 Conclusions 135
7.1 Thesis conclusions 137
7.2 Future work 139

Publications 139

List of Figures 143

List of Tables 147

Bibliography 149

1

Introduction

CHAPTER 1

INTRODUCTION

Many processors exploit Instruction Level Parallelism to execute several instructions from a single stream (thread) in parallel. However, there is only a limited amount of parallelism available in each thread due to data and control dependences, among other factors [68]:

- Control dependences: every time a control flow instruction changes the flow of instructions to a new target instruction, it takes several cycles to start fetching from that target, which degrades the number of instructions committed per cycle (IPC).
- Data dependences: data dependences limit the IPC as well since a instruction can only start its execution when all its input dependences are resolved. For short-latency operations the out-of-order mechanism of current superscalar processors hides part of this latency. However, when the processor experiences a long-latency operation, i.e., a miss in the outer cache level, this mechanism is not able to hide it causing a stall of the processor.

Processor designers use many hardware resources in order to reduce the effect of these problems, e.g., bigger and more complex branch predictors to control dependences and deeper windows to further exploit ILP when a long-latency instruction is executed. However, data and control dependences significantly limit performance, degrading the performance/cost ratio of processors.

A solution to improve the performance/cost ratio of processors is to allow threads to share hardware resources, like multithreaded (MT) processors do. There are several categories of MT processors, each dealing with the above problems in a different way. The classification of multithreaded processors is not well established. In this thesis, we have used a classification similar to that presented in [66][67], as explained in Figure 1.1. In this figure, A and B represent two different applications. White squares denote unutilized slots.

- In a super-scalar architecture, only one thread is running at a given time.
- In a multiprocessor, resources are not shared between threads [1]. Each thread uses a different set of resources.

[1] These applications likely share some levels of the cache hierarchy.

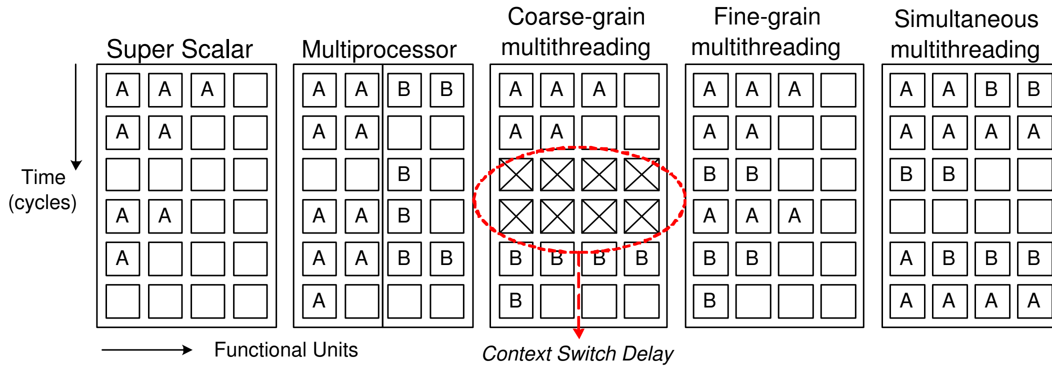


Fig. 1.1: A possible classification of multithreaded architectures

- In a coarse-grain multithreaded processor [6][61], threads share more execution resources than in a multiprocessor. Instructions can be issued from a single thread in a given cycle. A coarse-grain processor switches to a different thread when a thread experiences a long-latency operation, e.g., an outer cache miss. This allows the processors to hide part of the latency of long-latency operations.

- Fine-grain multithreading [8][29][39][58]: The main difference between coarse-grain and fine-grain multithreading is the granularity at which context switches occur. In a fine-grain multithreaded processor context switches are caused by other, not necessarily long-latency, events, e.g. branch misprediction. In this way, fine-grain processors can hide the latency of short-latency operations. Another difference between fine-grain and coarse-grain multithreading is that the latter approach switches between threads much more frequently than the former approach. As a result, in fine-grain multithreading context switches have lower cost (probably 1 cycle) than in coarse-grain multithreading.

- The main characteristic of simultaneous multithreading (SMT) processors [32][54][64][65][71] is their ability to issue instructions from the different threads in the same cycle. As a result, SMTs not only can they switch to a different thread to use the idle issue cycles in a short-latency operation (like fine-grain multithreaded), but also fill unused issue slots in a given cycle. Executing several threads at the same time provides a new form of parallelism, in addition to ILP, namely, thread level parallelism (TLP). This parallelism comes from the additional parallelism provided by the freedom to fetch instructions from different independent threads, and from mixing them in an appropriate way in the processor.

Regarding control dependences, MT processors reduce the dependence of throughput on branch prediction accuracy. That is, branch prediction accuracy is not of the utmost importance when running multithreaded applications [48][51].

This is mainly due to the fact that the opportunity of fetching from several threads reduces the percentage of speculative instructions on a wrong path [62].

Regarding data dependences, MT processors have shown to be successful in reducing the effect of data dependences [20][43][63]. This is due to the ability of MT processors to execute instructions from several threads [2].

Given all these advantages of MT processors, current trends in computer architecture show that many future processors will have some form of multithreading [7][41].

- High-performance systems: processors like the Intel Pentium4 Xeon [47], the IBM Power5 [36] and the SPARC64 VI [38] are multithreaded.
- Network systems: network multithreading processors are the Intel IXP network processor family [3] and the IBM PowerNP [4].
- Real-Time systems: the Imagination Technologies Meta processor [41] and the Infineon TriCore 2 [10] are two examples.

Each of these computing systems has different targets. For example, in an embedded real-time environment the target of the system could be to execute a given thread before a deadline while maximizing the overall system performance and reducing power consumption. On the other hand, in a high-performance system the target could be just to improve overall performance. Thus, the question is how a multithreaded processor could be designed to potentially meet all these requirements at the same time. We call these requirements *Quality of Service (QoS) requirements*.

1.1 SMT processors

Two are the main problems of SMT processors to meet quality of service requirements. First, SMTs are designed so that they intrinsically assume part of the responsibility of the system. The collaboration between the OS and the SMT is inherited from the traditional collaboration between the OS and multiprocessors: the OS perceives the different contexts of an SMT as multiple, independent *virtual processors*. As a result, the OS schedules threads onto what it regards as processing units operating in parallel. However, in an SMT, many hardware resources are shared between threads. Hence, these virtual processors are not truly independent as threads scheduled at any given time compete with each other for the resources of a single processor. The way that these resources are allocated, at the microarchitectural level, clearly affects their performance.

Second, current SMTs are designed with the main objective of increased throughput, lacking flexibility in providing other objectives. An instruction fetch (I-fetch) policy, e.g., *icount* [64], decides how instructions are fetched from the threads, thereby implicitly determining the way internal processor resources, like rename

[2] MT is orthogonal to the out-of-order mechanism of the processor, if it exists.

registers or issue queue (IQ) entries, are allocated to the threads. A common characteristic of many existing fetch policies is that they attempt to maximize throughput [64] and/or fairness [46][3], possibly by stalling or flushing threads that experience L2 misses [20][43][63], or reducing the effects of mispeculation by stalling on hard-to-predict branches [37][45].

Throughput or fairness might be acceptable objectives in many computing systems. However, the Operating System (OS) may need to impose additional objectives. This may relate only to some of the threads and may get priority over generic objectives such as fairness and/or throughput. For instance, in order to meet some real-time requirements, the OS may require that a designated thread runs at a certain percentage of the maximum speed it would get if it had the machine alone. When the designated thread is executed in an SMT processor, it will typically be sharing processor resources with other threads (if it was running on its own, it would clearly underutilize the SMT and defy the whole point about having SMTs). As a result, the designated thread's performance requirement cannot be guaranteed using the current collaboration between the OS and SMTs. This is because the fetch policy used for resource sharing would assign resources in order to meet its own overall objectives rather than those of the OS. If it would serve its purpose to allocate fewer resources to the designated thread, it would do so.

Even though SMT processors are good candidates for many future computing systems, these two problems (OS/SMT collaboration and the lack of flexibility), make the feasibility of SMTs for those systems difficult.

1.2 Thesis contributions

The main purpose of this thesis is to add hardware support in the SMT processor and software mechanisms in the OS with a two-fold objective. First, to improve the performance of SMTs in computing systems where SMTs are already used. Second, to allow SMTs to fulfill more QoS requirements and hence, to allow them to be used in a wider range of systems.

The main contribution of this thesis is that we propose for the first time the concept of *Explicit Resource Allocation* (ERA) for SMT processors as well as strategies that make use of this concept to provide flexible OS/SMT collaboration. We show that by directly controlling SMT shared resources we achieve our first objective, namely, improving SMT overall performance. Moreover, we explicitly specify to the OS how each thread uses resources. This enables SMTs to be used in other types of systems, our second objective. In this thesis, we focus on two main types of systems:

- High-performance systems: for high-performance systems we propose two I-fetch policies, called *Flush++* and *DWarn*.

[3] The concepts of throughput and fairness for SMTs are defined in the following chapters.

  - Flush++: the Flush++ policy improves previous I-fetch policies in the SMT literature in throughput and fairness while reducing the number of instructions flushed from the pipeline due to L2 data cache misses.
  - DWarn is a simple mechanism that provides good throughput/fairness trade-off, while removing the need of flushing instructions from the pipeline due to cache misses.

  These two mechanisms were the first step for the conception of the idea of explicit resource allocation. After that, and based on ERA we propose a third mechanism, *dcra*. Dcra moves one step further than I-Fetch policies, achieving a better throughput/fairness/power consumption balance than all evaluated fetch policies.

- Real-time systems: for real-time systems we propose two mechanisms, Predictable Performance and Low-Variability Performance, that in contrast to previous approaches, allow SMTs to be used in real-time environments. The main idea behind both proposals is to make explicit the use of resources to the OS. For that purpose, it is required small changes in the OS and hardware support in the SMT processor. Our results show that we achieve a success rate in meeting tasks deadline higher than 98% while the performance of the SMT is not heavily affected.

We would like to emphasize at this point that many of the quality of service requirements shown in this thesis were weakly defined or not defined for SMT processors at the time we analyzed them. The consequence of this was that, in some cases one of the major challenges we addressed was to redefine system requirements in an SMT environment.

1.3 Thesis structure

An intuitive organization of this thesis would be to start defining the concept of Quality of Service for SMT processors and next, showing our hardware/software changes to provide quality of service on each of the two systems discussed in the previous section. The main problem of this organization is that the concept of quality of service we originally applied to SMTs changed as this thesis was being developed. Furthermore, some requirements were addressed differently before and after the concept of QoS was defined. For this reason, we will use a chronological order to show our proposals. We show the quality of service requirements as well as our proposals in the same order we addressed them.

This thesis is structured as follows:

- Chapter 2 is devoted to explain our experimental environment. This includes both the simulation tools and the benchmarks used in this thesis.

- Chapter 3 defines the requirements that high-performance systems can have on SMT processors. This section also shows our first two proposals to meet these requirements.
- Chapter 4 describes our concept of quality of service for SMT processors as well as the catalyst of that concept, namely, *explicit resource allocation*. We show how by explicitly controlling the resource allocation in an SMT, we can move one step further than previous proposals. On the one hand, we improve those techniques focused on high-performance systems. On the other hand, we enable SMT processors to be used in other type of systems, like real time systems.
- Chapter 5 presents the last of our proposals focused on high-performance systems. This last proposal is based on the concept of explicit resource allocation.
- Chapter 6 is devoted to present the studies we did on SMT processors for real-time systems. We analyze in this chapter the requirements of a real-time system, previous approaches, and finally, we show our own proposals.
- Chapter 7 shows the conclusions of this thesis.

2

Experimental Framework

CHAPTER 2

EXPERIMENTAL FRAMEWORK

This chapter is devoted to explain the evaluation tools we have used in order to check our proposals. We show the benchmarks used for that purpose as well.

2.1 Evaluation tool

To evaluate the performance of the different mechanisms shown in this thesis, we use a trace driven SMT simulator derived from *smtsim* [65]. 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.

In our baseline architecture, shown in Figure 2.1, the fetch logic fetches instructions from the instruction cache in program order. We have used two fetch architectures.

- 1.X: up to X instructions can be fetched each cycle from a single thread.
- 2.X: up to X instructions can be fetched each cycle from at most two threads.

An instruction fetch policy determines from which of the available threads instructions are fetched. Next, instructions are decoded and renamed in order to track data dependences. When an instruction is renamed, it is allocated an entry in the issue queues (window or IQs) until all its operands are ready. Each instruction also allocates one Re-Order Buffer (ROB) entry and a physical register, if required. ROB entries are assigned in program order. When an instruction has all its operands ready, it is issued[1]: it reads its operands, executes, writes its results, and finally commits.

The data and the instruction caches are accessed with physical addresses. The data cache uses write back as write hit policy and write allocate as write miss policy [35]. Caches are tagged with the identifier of threads so that threads do not share data and/or instructions.

A key parameter for all the studies discussed in this thesis are the resources that are shared between threads and which are exclusive for each thread. The main shared resources in our baseline architecture are the following:

[1] In this thesis, the term *dispatch* indicates the action of moving instructions to the issue queues. The term *issue* is applied to the action of submitting instructions from the issue queues to the backend of the machine.

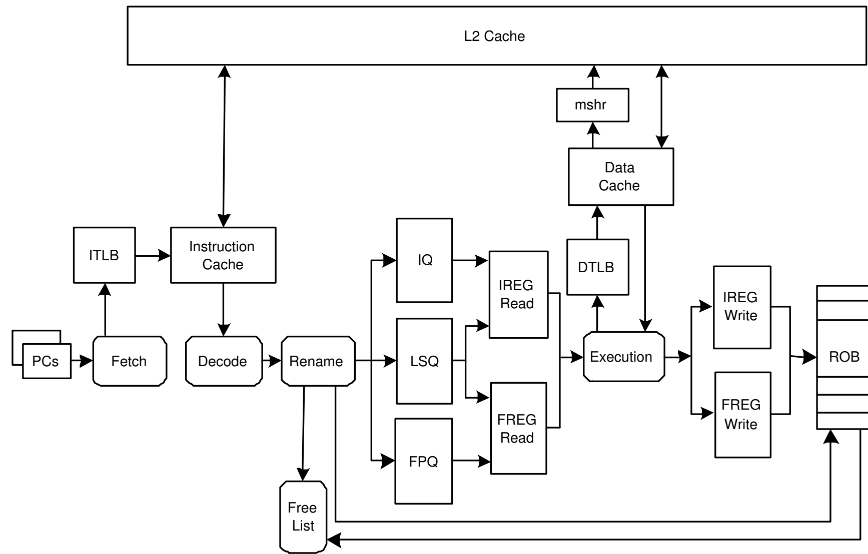


Fig. 2.1: Block Diagram of our baseline architecture

- The fetch bandwidth: this is one of the key parameters for the performance of the processor [64]. The instruction fetch policy determines which threads use the fetch bandwidth.

- The issue queues: after the rename stage the processor considers all instructions equal until they are committed. The IQs are shared between all threads.

- The issue bandwidth: unless stated otherwise, the default issue policy we use is *first in first out* (oldest first).

- The physical registers: like the issue queues, the physical register file is common to all contexts.

- Caches (the instruction cache, the L1 data cache, and the unified L2 cache): caches are shared between threads. However, caches are tagged with the identifier of the context. Hence, the data of one context is not shared with any other context.

- TLBs (the instruction TLB and the data TLB): like caches, TLBs are tagged with thread identifier. In Chapter 6 it is shown that the number of misses of the threads in the TLBs is low. For this reason, TLBs are not taken into account throughout this thesis.

- Reorder Buffer: in some of our experiments we use a shared ROB while in other we use a separated ROB for each thread. This only slightly changes performance results as long as we control physical registers as well. That is, if we control the physical registers, the influence of the ROB is not significant in our architecture.

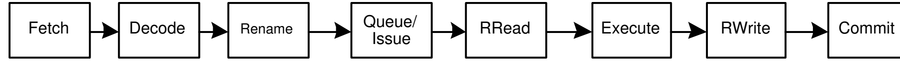


Fig. 2.2: Stages in our baseline architecture

The pipeline of our baseline architecture is composed of eight stages as shown in Figure 2.2. In some of the experiments shown in the following chapters, we consider that access to the register file (*RRead* and *RWrite* stages) takes two cycles. In other experiments, the *Decode* stage takes up to 5 cycles and the *Rename* stage up to 3 cycles. In this way, the number of stages of our pipeline varies from 8 up to 16.

2.2 Benchmarks

We use independent[2] programs from the SPEC2000 integer and fp benchmark suite shown in Table 2.1.

Table 2.1: FastForward used for each Spec CPU 2000 Benchmark.

Benchmark type	Benchmark name	Input	Language	Fast Forward (Millions)
INTEGER	164.gzip	graphic	C	68.100
	175.vpr	place	C	2.100
	176.gcc	166.i	C	14.000
	181.mcf	inp.in	C	43.500
	186.crafty	crafty.in	C	74.700
	191.parser	ref.in	C	83.100
	252.eon	cook	C++	57.600
	253.perlbmk	splitmail.535	C	45.300
	254.gap	ref.in	C	79.800
	255.vortex	lendian1.raw	C	58.200
	256.bzip2	inp.program	C	13.500
	300.twolf	ref	C	324.300
FP	168.wupwise	wupwise.in	Fortran77	263.100
	171.swim	swim.in	Fortran77	47.100
	172.mgrid	mgrid.in	Fortran77	187.800
	173.applu	applu.in	Fortran77	10.200
	177.mesa	frames100 + msea.in	C	294.600
	178.galgel	gagel.in	Fortran90	175.800
	179.art	-scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10	C	13.200
	183.equake	inp.in	C	27.000
	188.ammp	ammp.in	C	13.200
	189.lucas	lucas2.in	Fortran90	30.000
	191.fma3d	fma3d.in	Fortran90	10.500
	301.apsi	apsi.in	Fortran77	192.600

Each program is executed using the reference input data set and compiled with the $-O2-non_shared$ options using the DEC Alpha AXP-21264 C/C++ compiler. Fortran programs are compiled with the DIGITAL Fortran 90/Fortran 77 compilers

[2] By independent we mean non-parallel programs.

with the $O2 - g3 - non_shared$ options. Traces of the benchmarks are collected of the most representative 300 million instruction segment, following an idea presented in [22][56].

Programs are divided into two groups based on their cache behavior, as shown in Table 2.2. For our experiments, we use an L2 cache of 512 Kb. Those benchmarks with an L2 cache miss rate higher than 1% are considered memory bounded (MEM), and the other are considered ILP (CPU bounded). The L2 miss rate is calculated with respect to the number of dynamic loads.

Table 2.2: Cache behavior of each benchmark in a 512Kb L2 cache

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) Memory bounded benchmarks

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) CPU bounded benchmarks

In the final part of this thesis, we use real-time applications drawn from ten voice, image and video coder and decoder applications from the MediaBench suite [1][40]. Table 2.3 shows the applications used in this thesis. MediaBench programs are executed until completion.

Table 2.3: MediaBench Benchmarks used in this thesis.

Benchmark name	Media	Language	input
adpcm	speech	C	clinton.pcm
epic	image	C	test_image.pgm
gsm	speech	C	clinton.pcm
g721	speech	C	clinton.pcm
mpeg2	video	C	test2.mpeg

3

QoS for High-Performance Systems with Implicit Resource Allocation

CHAPTER 3

QOS FOR HIGH-PERFORMANCE SYSTEMS WITH IMPLICIT RESOURCE ALLOCATION

In this chapter we analyze the requirements that a high-performance system can request from an SMT processor. As we will see, these requirements are mainly throughput and fairness. Next, we analyze the state-of-the-art proposals. Finally, we show our first two proposals: Flush++ (Section 3.4) and DWarn (Section 3.5).

3.1 Introduction

It has been demonstrated that Multithreaded and Simultaneous multithreaded architectures are promising means of achieving high performance by better using processor resources [32][54][64][65][71], with a moderate area overhead over a superscalar processor [12][36][47]. In an SMT processor, several threads run together with the objective of increasing available instruction level parallelism. Co-scheduled threads use exclusively some machine resources, like the reorder buffer, and share others like the issue queues or the physical registers, the functional units, and the physical registers. Shared resources are dynamically allocated between threads competing for them. An I-fetch policy determines which threads enter the processor, and hence, how shared resources are allocated.

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 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 over 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 should be taken into account.

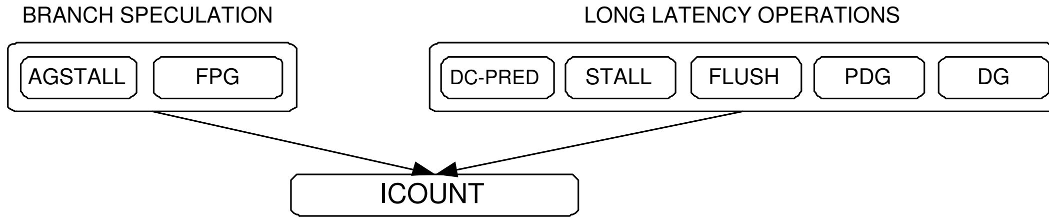


Fig. 3.1: Schema of the related policies

3.2 Related work

Current approaches to improve the throughput and fairness include both software and hardware approaches. Software approaches focus on compiler techniques [44][49] and operating system techniques [55] that try to reduce the interference in SMT hardware shared resources. In this thesis we focus on hardware approaches. The main hardware approaches are explained next.

In [64] it is observed that the throughput of an SMT processor is quite sensitive to the I-Fetch policy. In [64] several fetch policies are proposed, from which we explain the main two. First, *round-robin* [64] is the most basic form of fetch and simply fetches instructions from all threads alternatively, disregarding the resource use of each thread. Second, *icount* [64] prioritizes threads with few instructions in the issue queues (and in the pre-issue stages). From the policies presented in [64] icount provides the best performance results.

The icount policy presents good results for threads with high ILP. However, SMTs have special problems, not addressed by either round-robin or icount, that may cause a significant performance drop. These problems are two:

- Branch mispredictions: resources and power are wasted by executing instructions along a mispredicted path, and icount assigns priorities disregarding the fact that a thread may be on a wrong speculative path. Some current policies, like *Fetch Prioritizing and Gating* (fpg) [45], and *AGstall* [37] work on top of icount (see Figure 3.1), and add branch speculation control to reduce this waste. These policies obtain a significant reduction in waste and a slight improvement in total IPC.

- Load miss latency: icount and round-robin also disregard that a thread can be blocked on an L2 miss, and will not make progress for many cycles. The icount policy only takes into account the occupancy of the issue queues, giving higher priority to those threads with fewer instructions in the queues (and in the pre-issue stages). When a load misses in L2, dependent instructions occupy the issue queues for a long time.

  - On the one hand, if the number of dependent instructions is high, this thread will have low priority. However, these entries cannot be used by the other threads degrading their performance.

- On the other hand, if the number of dependent instructions after a load missing in L2 is low, the number of instructions in the queues is also low. Hence this thread will have high priority and will execute many instructions that cannot be committed for a long time. As a result, the processor can run out of registers.

Therefore, the icount policy only has a limited control over the issue queues and ignores the occupancy of the physical registers.

In this chapter we focus on policies dealing with the problem of long latency loads. The performance of these type of policies depends on the following two factors: the detection moment (DM) and the response action (RA).

- The DM indicates the moment in which a policy detects a load that fails or is predicted to fail in cache. Possible values range from the cycle the load is fetched into the processor until the cycle in which the load finally fails in the L2 cache. Two characteristics associated with the DM are the *reliability* and the *speed*. The higher the speed of a method to detect a delinquent load is, the lower its reliability. On the one hand, if we wait until the load misses in L2, we know for certain that it is a delinquent load: totally reliable but too late. On the other hand, we can predict which loads are going to miss by adding a load miss predictor to the front-end [43]. In this case, the speed is higher, but the reliability is low due to predictor misses.

- The RA indicates the behavior of the policy once a load is detected or predicted to miss in cache, that is, it defines the measures that the fetch policy takes for delinquent threads.

With these two parameters, we classify all current policies related to long latency loads, see Table 3.1.

In [43], a mechanism called *dc-pred* is presented. It uses a load miss predictor to detect the L2 missing loads in the fetch stage (we call this DM $fetch$) and for the RA, the corresponding thread is restricted to use a maximum amount of available resources ($limit\ resources$). When the missing load is resolved the thread is allowed to use all resources. The main problem of this policy is that the $fetch$ DM does not detect all loads missing in L2 and hence some loads that actually fail in the cache and that are not predicted to miss, can clog the shared resources.

In [63] two main mechanisms are proposed, both using the $x\ cycles\ after\ issue$ DM. When this DM is used, a load is declared to miss in the L2 cache when it spends more cycles in the cache hierarchy than needed to access the L2 cache, including possible resource conflicts. The first mechanism stalls the delinquent thread until the the load is resolved (*gate* RA). We call this combination *stall*. The main problem of the stall policy is that the L2 miss detection already may be too late to prevent a thread from occupying most of the available resources. For this reason authors propose another mechanism that, in addition to stall the offending thread as long as

Table 3.1: Response Action \ Detection Moment space

RA\DM	Fetch cycle	L1 Data cache	X cycles after issue	L2 cache
gate	pdg	dg	stall	
squash			flush	
limit resources	dc-pred			

the load is not resolved, it flushes the instructions of the delinquent thread after the L2 missing load (*squash* RA). We call this combination *flush*. As a result of applying flush, the offending thread temporarily does not compete for resources and, more importantly, the resources used by this thread are freed, giving the other threads full access to them. Stall is less aggressive than flush, does not require hardware as complex as flush, and does not re-execute instructions. However, its performance results are worse.

In [20], two mechanisms are proposed to reduce clogs in the issue queues caused by loads that miss in the L1 data cache. The first mechanism, *data gating*, or *dg*, stalls a thread when it has more than n outstanding misses. The second one, *predictive data gating* or *pdg*, is more aggressive than dg and adds an L1 data cache miss predictor. The pdg policy stalls a thread when the number of loads predicted to miss plus the number of loads predicted to hit that in reality miss, is higher than n. In [20] the authors use the value $n = 0$, hence with dg a thread is stalled on each L1 miss. That is, it uses the $l1$ DM and the *gate* RA. With pdg threads are stalled on each predicted miss ($fetch$ DM and *gate* RA). The main problem of the dg and pdg policies arises when there are few threads, because the exposed parallelism and the pressure on resources are low. Consequently, to stall a thread every time it has an L1 data miss may cause an under-utilization of the resources.

3.3 Metrics in high-performance systems

Several performance metrics have been proposed for SMT processors. Some of these metrics try to balance throughput and fairness [46]. 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 [46], as it has been shown that it offers better fairness-throughput balance than $Weighted\ Speedup$ [53][63]. 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.

We call the fraction IPC_{SMT}/IPC_{alone} the *relative IPC*, where the IPC_{SMT} is the IPC of a thread in a given workload, and the IPC_{alone} is the IPC of a thread when it runs in isolation. The Hmean metric is the harmonic mean of the relative IPC of the threads in a workload. *Hmean* is calculated as shown in equation 3.1.

$$Hmean = \frac{\#threads}{\sum_{threads} \frac{IPC_{alone}}{IPC_{SMT}}} \tag{3.1}$$

Finally, we use the *extra fetch* (EF) metric. It measures the extra instructions fetched due to flushing of instructions, see equation 3.2. Here we are not taking into account the flushed instructions due to branch misspredictions, but only those related with the loads missing in L2. EF compares the total fetched instructions (flushed and not flushed) with the instructions that are fetched and not flushed. The higher the value of the EF the higher the number of squashed instructions respect to the total fetched instructions. If no instructions is squashed, EF is equal to zero.

$$EF = \left(\frac{TotalFetched}{Fetched\ not\ squashed} - 1\right) * 100(\%) \tag{3.2}$$

With these three metrics we measure the efficiency of all our proposals for high-performance systems.

3.4 Flush++

When a thread experiences an L2 cache miss, instructions after this load occupy resources for a long time while making little progress. Each instruction occupies a ROB entry and many of them a physical register from the rename stage to the commit stage. It also uses an entry in the issue queues while any of its operands is not ready, and a functional unit (FU). Neither the ROB nor the FUs present a problem, because the ROB is not shared and the FUs are pipelined. The issue queues and the physical registers are the actual problem, because they are used for a variable, long period. Thus, the instruction fetch policy must prevent an incorrect use of these shared resources to avoid significant performance degradation.

Several policies have been proposed to alleviate the previous problem as shown in the related work section of this chapter. In the first part of this section we compare several of these fetch policies. Next, we analyze the use of a load miss predictor in an SMT processor, and compare it with the flush and stall policies. As we show below, no policy outperforms all others in all cases: each behaves better than the others depending on the metric used and on the workload. Based on this initial

study, in the second part we propose improved versions of each policy. Throughput and fairness [46] results show that in general improved versions achieve significant performance improvement over the original versions for a wide range of workloads, ranging from two to eight threads.

3.4.1 Experimental setup

As show in Chapter 2, we use a trace driven SMT simulator, based on smtsim [65]. The main parameters of our baseline configuration is shown in Table 3.2.

Table 3.2: Baseline configuration

Processor Configuration	
Fetch/Issue/Commit Width	8
Baseline fetch policy	icount
Baseline fetch architecture	2.8
Issue Queue Entries	32 int, 32 fp, 32 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	384 int, 384 fp
ROB size	256 entries per thread
Branch Prediction Configuration	
Branch Predictor	2K 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, 2-way, 8-bank, 64-byte lines, 10 cycle access
Main memory latency	100 cycles
TLB miss penalty	160 cycles

From SPEC benchmarks and based on their cache behavior, Figure 2.2, we have created 12 workloads, as shown in Table 3.3, ranging from 2 to 8 threads. In the ILP workloads, all benchmarks have good cache behavior. All benchmarks in the MEM workloads have an L2 miss rate higher than 1%. Finally, the MIX workloads include ILP threads as well as MEM threads. For MEM workloads some benchmarks were used twice, because there are not enough SPECINT benchmarks with bad cache behavior. The replicated benchmarks are highlighted in boldface in Table 3.3. We have shifted the second instance of a replicated benchmarks by one million instructions in order to avoid that both threads accessing the cache hierarchy at the same time.

3.4.2 Load miss predictors in an SMT environment

We have analyzed the performance of several load miss predictors. Before explaining each of these predictors, we describe hardware needed to enable the use of load miss predictors, which we call *l2mp*. This policy is similar to pdg [20], but it predicts L2

Table 3.3: Workload classification based on cache behavior of threads.

# of threads	Workload type	Workloads
2	ILP	gzip, bzip2
	MIX	gzip, twolf
	MEM	mcf, twolf
4	ILP	gzip, bzip2, eon, gcc
	MIX	gzip, twolf , bzip2, mcf
	MEM	mcf, twolf, vpr, **twolf**
6	ILP	gzip, bzip2, eon, gcc crafty, perlbmk
	MIX	gzip, twolf, bzip2, mcf, vpr, eon
	MEM	mcf, twolf, vpr, parser, **mcf**, **twolf**
8	ILP	gzip, bzip2, eon, gcc crafty, perlbmk, gap, vortex
	MIX	gzip, twolf, bzip2, mcf, vpr, eon, parser, gap
	MEM	mcf, twolf, vpr, parser, **mcf**, **twolf**, **vpr**, **parser**

misses instead of L1 misses. After that, we describe the load miss predictors that we evaluated as well as the metrics used to compare the effectiveness of the different load miss predictors.

The l2mp policy

The l2mp scheme is shown in Figure 3.2. The predictor acts in the decode stage. It is a table indexed by the PC of a load: if a load is not predicted (1) to miss in L2, it executes normally. If a load is predicted (1) to miss in L2 cache, the thread it belongs to is stalled (2). This load is tagged indicating that it has stalled the thread. When this load is resolved, either in the Dcache (3), or in the L2 cache(4), the corresponding thread resumes.

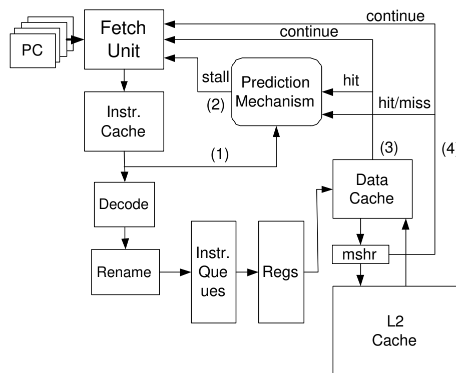


Fig. 3.2: l2mp mechanism

Load miss predictors

We have explored a wide range of load miss predictors. The one that obtains the best results is the predictor proposed in [43]. We call this predictor *pattern predictor*.

In this section, we show the results of the pattern predictor and also the results of the predictor proposed in [20], called *2bc*.

The pattern predictor uses one table per thread. Each table is direct mapped and each of its entries contains three fields: a tag, the number of hits in the cache (Data cache and L2 cache) between the last two misses in L2, and the number of hits in the cache since the last miss. A load is predicted to miss in L2 when the last two fields are equal. The table is updated when a load hits in the cache (Data cache or L2) and when the load misses in L2.

The 2bc predictor uses only one table that is shared between threads. This table is indexed by the PC of the load and it has 2K entries of two bit saturating counters. On a miss, the corresponding entry is cleared and on a hit, it is incremented. The most significant bit determines the prediction.

Metrics to compare load miss predictors

As in [72], we classify every dynamic load into one of four groups based on the result of the load (hit or miss in cache), and on the prediction made by the predictor. The contribution of this section is that we identify and analyze the particularities of each group in an SMT processor.

We differentiate four groups, the first two represent predictor hits, and the last two predictor misses.

- AHPH (actual hit - predicted hit): this group is formed by the loads that hit in the Data cache or in the L2 cache and that are not predicted to miss. Thus, they are executed normally.

- AMPM (actual miss - predicted miss): this group represents the loads missing in the L2 cache that are detected by the predictor. That is, the loads correctly stalled. This improves performance as we detect loads that are going to miss in the L2 cache and clog resources.

- AHPM (actual hit - predicted miss): loads that hit in the L1 cache or in the L2 cache that are predicted to miss. In this case, the corresponding thread is unnecessarily fetch-stalled until this load is resolved. Hence, its performance is degraded. If the load hits in L1 data cache, it takes approximately 5 cycles in the simulated architecture to re-start the thread. If the load miss in L1 and hits in L2, it takes approximately 15 cycles in the simulated architecture.

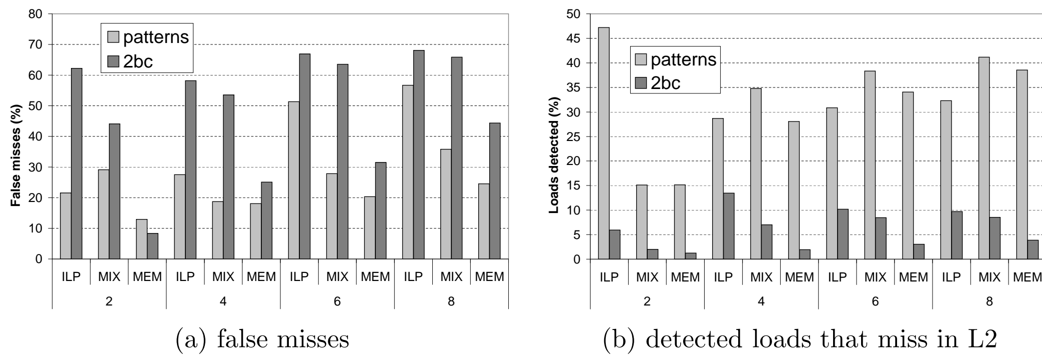
- AMPH (actual miss - predicted hit): this group covers those loads that actually miss in L2 but that are not detected by the predictor. These loads heavily degrade the performance of the SMT processor because machine resources are clogged by the instructions after these loads.

To measure the effectiveness of the predictors we use two metrics. First, *false misses*: the percentage of incorrect mispredictions (AHPM/PM). Second, *filtered*

loads: the percentage of actual misses (AM) that are detected by the predictor (AMPM/AM). The objective of the predictor is to maximize the filtered loads while minimizing the false misses. However, there is a trade-off between these two factors, because to achieve a high percentage of filtered loads the predictor must be more aggressive probably increasing the false misses.

Load miss predictors evaluation

Regarding to the pattern predictor, in [43] authors do not indicate the size of the predictor. We have explored different sizes, and the best result is obtained when the table has 8 entries. In this case the total predictor size is 2.944 Kbits (8 threads x 8 entries x (30 bits of tag + 8 bit counter + 8 bit counter)). Figures 3.3(a) and (b) compare the effectiveness of both predictors. These figures respectively show the percentage of false misses and the percentage of detected loads missing in L2. We observe that the pattern predictor presents the best results: it has less false misses and detects more loads loads that miss in the L2 cache. Hence, we use this predictor in the l2mp policy in the remainder of this section.

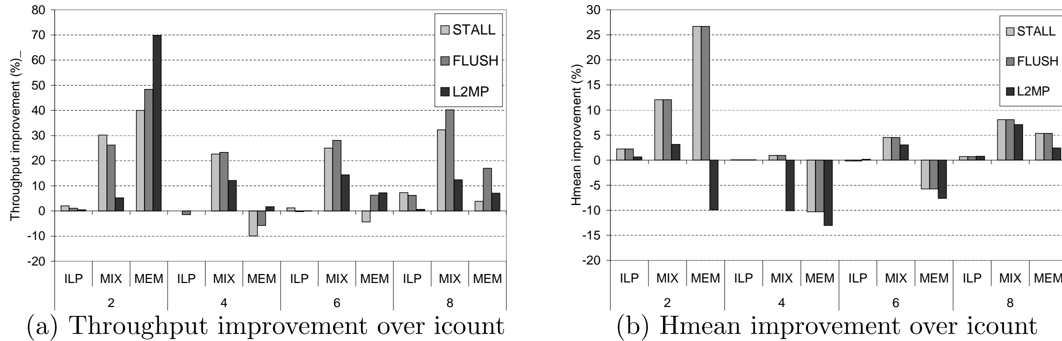


(a) false misses　　　(b) detected loads that miss in L2

Fig. 3.3: Effectiveness of the pattern predictor and the 2bc

3.4.3 Comparing the current policies

In this section we determine the effectiveness of the different policies addressing the problem of load latency. We compare the stall, flush and l2mp policies using the throughput and the Hmean metrics. In Figure 3.4, we show the throughput and the Hmean improvements of stall, flush, and l2mp over icount. L2mp achieves important throughput improvements over icount, mainly for 2-thread workloads. However, fairness results using the Hmean metric indicate that for the MEM workloads the l2mp is more unfair than icount. Only for 8-thread workloads l2mp outperforms icount in both throughput and Hmean. Our results indicate that this is because l2mp hurts MEM threads and boosts ILP threads, especially for few-thread workloads. If we compare the effectiveness of l2mp to other policies addressing the same problem, like stall, we observe that l2mp only achieves better throughput than stall for MEM



(a) Throughput improvement over icount (b) Hmean improvement over icount

Fig. 3.4: Comparing current policies

workloads. However, l2mp heavily affects fairness. We explain why l2mp does not obtain results as good as stall below.

The results of flush and stall are very similar. In general flush slightly outperforms stall, especially for MEM workloads and when the number of threads increases. This is because when the pressure on resources is high it is more preferable to flush delinquent threads and hence free resources, rather than stall these threads which causes them to hold resources for a long time. As stated before, no policy outperforms all others either for all workloads or for all metrics. Each one behaves better than others depending on the particular metric and workload.

3.4.4 Improved policies

In this section we present our improvements of the policies discussed previously: l2mp, stall y flush. Finally, based on the behavior of all these policies we propose a new policy called Flush++.

Improving l2mp

We have seen that l2mp alleviates the problem of load latency, but it does not achieve results as good as other policies addressing the same problem. The main drawback of l2mp is the AMPH loads, that is, those loads missing in the L2 cache that are not detected by the predictor. These loads can seriously damage performance because subsequent instructions occupy resources for a long time. Figure 3.5 indicates that this percentage is quite significant from 50% to 80%, and thus the problem still persists. We propose to add a safeguard mechanism to “filter” these undetected loads. That is, a mechanism that acts on loads missing in L2 that are not detected by the predictor. The objective is to reduce the harmful effects caused by these loads. In this section, we use stall [63] as a safeguard mechanism.

Our results show that, when using l2mp, the fetch is totally idle for many cycles (15% for the 2-MEM workload) because all threads are stalled by the l2mp mechanism. Another important modification that we have made to the original l2mp

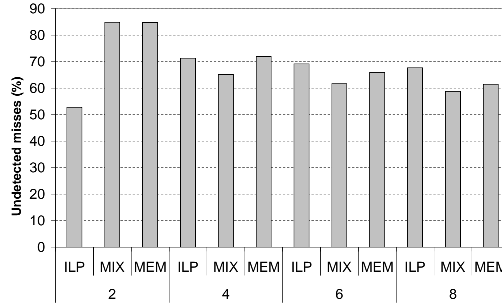
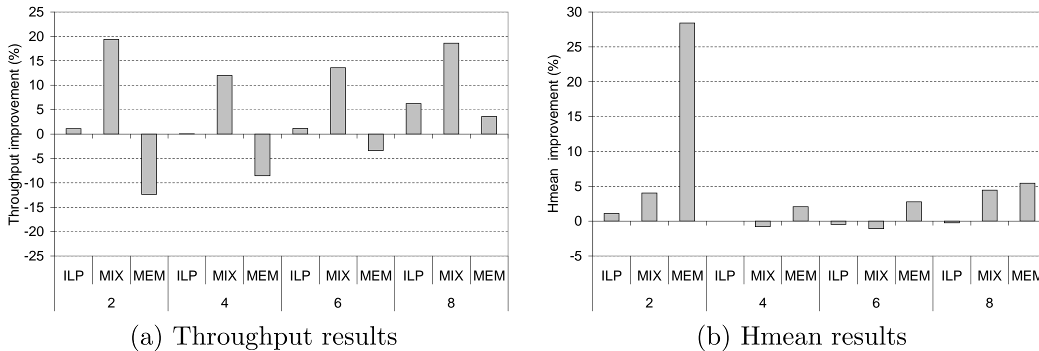


Fig. 3.5: Undetected L2 misses when the pattern predictor is used

mechanism, is to always keep one thread running in order to avoid idle cycles of the processor. We call the resulting policy l2mp+.

Figures 3.6 (a) and (b) show the throughput and the Hmean improvement of l2mp+ over l2mp. Throughput results show that for MEM workloads l2mp outperforms l2mp+ by 5.1% on average, and for MIX l2mp+ outperforms l2mp by 16% on average. We have investigated why l2mp improves l2mp+ for MEM workloads. We detected that l2mp+ significantly improves the IPC of *mcf* (a thread with a high L2 miss rate), but this causes an important reduction in the IPC of the remaining threads. Given that the IPC of *mcf* is very low, the decrease in IPC of the remaining threads affects the overall throughput more than the improvement in IPC of *mcf*. Hmean results, see Figure 3.6 (b), confirm that the l2mp+ policy presents a better throughput/fairness balance than the l2mp policy, only suffering slowdowns less than 1%.



(a) Throughput results

(b) Hmean results

Fig. 3.6: Improvement of the l2mp+ policy over the l2mp policy

Improving flush

The flush policy always attempts to leave one thread running. In doing so, it does not flush and fetch-stall a thread if all remaining threads are already fetch-stalled. Figure 3.7 shows a timing example for 2 threads. In cycle c0, thread T0 experiences

an L2 miss and it is flushed and fetch-stalled. After that, in cycle c1, thread T1 also experiences an L2 miss, but it is not stalled because it is the only thread running. The main problem of this policy is that by the time the missing load of T0 is resolved in cycle c2 and this thread can proceed, the machine is presumably filled with instructions from thread T1. These instructions occupy resources until the missing load of T1 is resolved in cycle c3. Hence, performance is degraded.

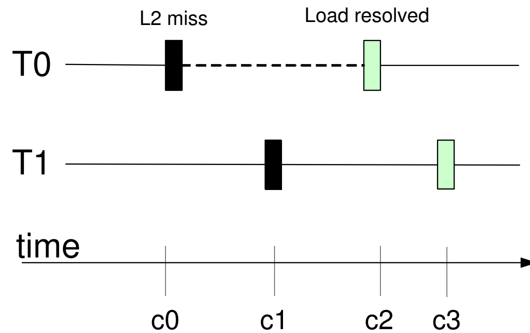


Fig. 3.7: Timing of the flush policy

The improvement we propose is called Continue the Oldest Thread (COT). When there are N threads, $N-1$ of them are already stalled, and the only thread running experiences an L2 miss, it is stalled and flushed, but the thread that was first stalled is continued. For the previous example, the new timing is depicted in Figure 3.8. When thread T1 experiences an L2 miss, it is flushed and stalled and T0 is continued. Hence, instructions from T0 consume resources until cycle c2 when the missing load is resolved. However, this does not affect thread T1 because it is stalled until cycle c3. In this example, the COT improvement has been applied to flush, but it can be applied to any fetch-gating policy. We have applied it also to stall. We call the new versions of flush and stall, flush+ and stall+, respectively.

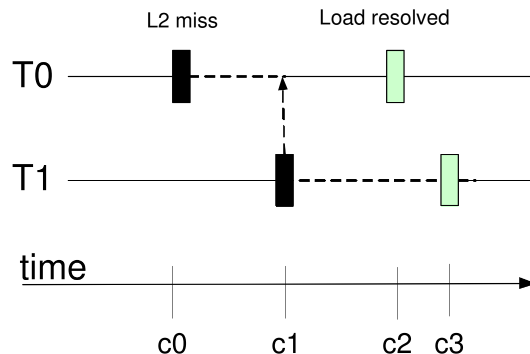
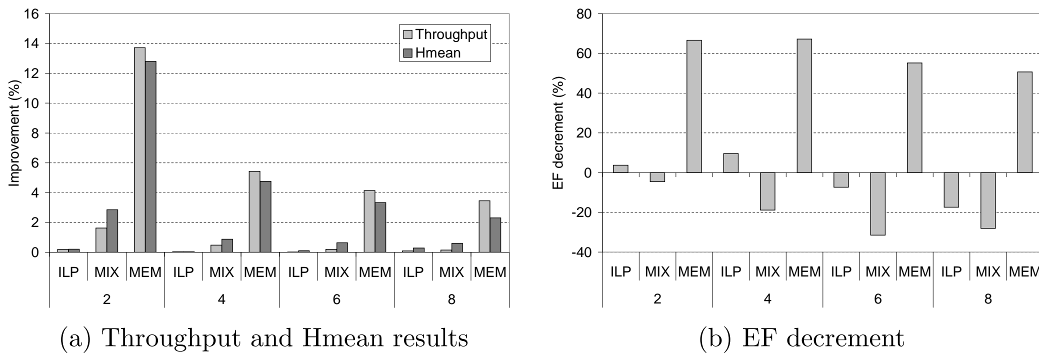


Fig. 3.8: Timing of the improved flush policy

Figure 3.9 (a) shows the throughput and the Hmean improvements of flush+ over flush. We observe that flush+ improves flush for all workloads. We also observe

that for MEM workloads flush+ clearly outperforms flush, for both metrics, and this improvement decreases as the number of threads increases. This is because, as the number of threads increases, the number of times that only one thread is running and the remaining are stopped, is lower. For MIX and ILP workloads, the improvement is lower than for the MEM workloads because the previous situation is less frequent. Regarding flushed instructions, in Figure 3.9(b) we see the EF decrement of flush+ over flush . We observe that, on average, flush+ reduces EF by 60% for MEM workloads compared to flush, and only increases EF by 20% for MIX workloads. These results indicate that flush+ presents a better throughput/fairness balance than flush, and moreover reduces extra fetch.



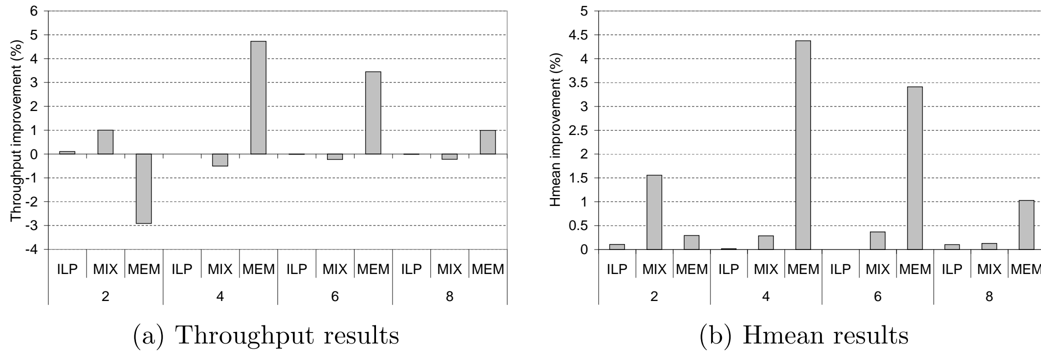
(a) Throughput and Hmean results (b) EF decrement

Fig. 3.9: Improvement of the flush+ policy over the flush policy.

Improving stall

Figures 3.10 (a) and (b) show the throughput and the Hmean improvement of stall+ over stall. We observe that the improvements of stall+ over stall are less pronounced than the improvements of flush+ over flush. Throughput results show that in general stall+ improves stall, and only for the 2-MEM workload there is a remarkable slowdown of 3%. The Hmean results show that stall+ outperforms stall for all workloads, and especially for MEM workloads. We do not show EF results because the stall and stall+ policies do not squash instructions.

We have analyzed why stall outperforms stall+ for the 2-MEM workload. We observed that this is caused by the benchmark mcf. The main characteristic of this benchmark is its high L2 miss rate. On average, one of every eight instructions is a load failing in L2. In this case, the COT improvement behaves as show in Figure 3.11: in cycle c0 the thread T0 (mcf) experiences an L2 miss and it is fetch-stalled. After that, in cycle c1, T1 experiences an L2 miss it is stalled and T0 (mcf) is continued. A few cycles after that, mcf experiences another L2 miss, thus control is returned to thread T1. With flush, every time a thread is stalled it is also flushed. With stall, this is not the case. Hence, from cycle c1 to c2, mcf allocates resources that are not freed for a long time, degrading the performance of T1. That is, the COT improvement for the stall policy improves the IPC of benchmarks with high



(a) Throughput results (b) Hmean results

Fig. 3.10: Improvement of the stall+ policy over the stall policy.

L2 miss rate, *mcf* in this case, but hurts the IPC of the remaining threads. This situation is especially acute for 2-MEM workloads. To solve this problem, and other ones, we develop a new policy called flush++.

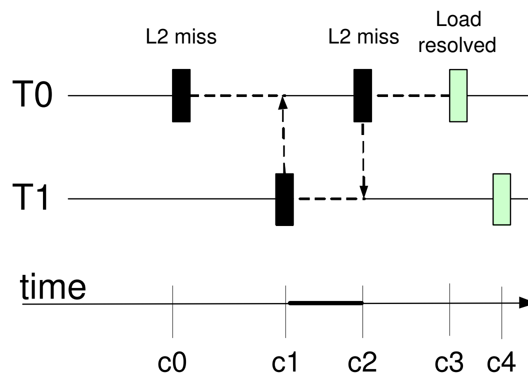


Fig. 3.11: Timing when a thread with high L2 miss rate is executed

3.4.5 The flush++ policy

The flush++ policy tries to combine the advantages of both previous policies, stall and flush+. It focuses in the following three points.

- First, for MIX workloads stall presents good results. It is an alternative to flush+ avoiding instruction re-execution.
- Second, another objective is to improve the IPC of stall for MEM workloads with a moderate increment in the re-executed instructions.
- Third, the processor knows every cycle the number of threads that are running. This information can be easily obtained for any policy at runtime.

Flush++ works differently depending on the number of running threads. If the number of running threads is less than four, it combines stall and flush+. It behaves

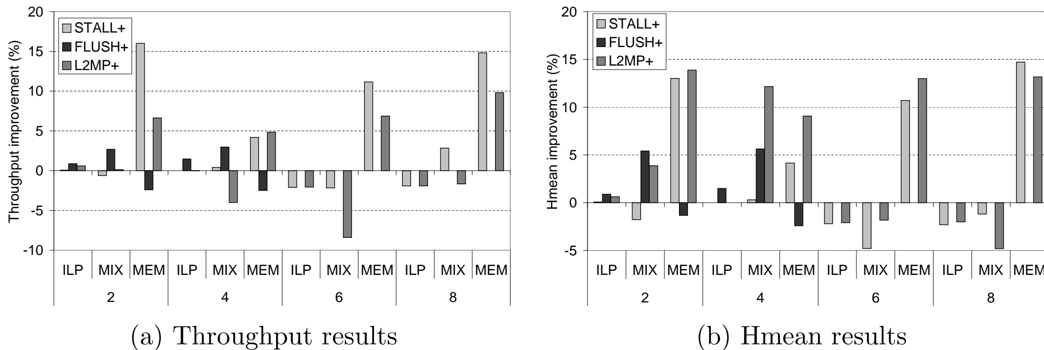
like stall but when the COT improvement is triggered it acts as flush+. That is, the flush is only activated when there is only one thread running and it experiences an L2 miss. In the remaining situations, threads are only stalled. When there are more than four threads running, we must consider two factors. On the one hand, the pressure on resources is high. In this situation is preferable to flush delinquent threads instead of stalling them because freed resources are highly profitable for the other threads. On the other hand, flush+ improves flush in both throughput and fairness for four-or-more thread workloads. For this reason, if the number of threads is greater than four, we use the flush+ policy.

In Figure 3.12 we compare flush++ with the improved versions of existing fetch policies: stall+, l2mp+, and flush+. Figure 3.12(a) shows the throughput results and Figure 3.12(b) the Hmean results.

Regarding l2mp+, throughput results show that l2mp+ improves flush++ for MIX workloads, and that flush++ is better than l2mp+ for MEM workloads. The Hmean results indicate that only for 6-, and 8-thread workloads l2mp+ is slightly more fair than flush++ for ILP and MIX workloads. For the remaining workloads flush++ is more fair. In general, flush++ outperforms l2mp+.

For ILP and MIX workloads flush++ outperforms flush+ and for MEM workloads it is slightly worse. The most interesting point is that, as we can see in Figure 3.13, flush++ considerably reduces the EF of flush+. For 6-, and 8-thread workloads the results are the same as those for flush+.

We observe that on average flush++ outperforms all other policies in both throughput and fairness.



(a) Throughput results (b) Hmean results

Fig. 3.12: Improvement of the flush++ policy over the flush, stall, flush+ and stall+ policies.

3.4.6 Conclusions

SMT performance directly depends on how the allocation of shared resources is done. The instruction fetch mechanism dynamically determines how this allocation is carried out. To achieve high performance, it must avoid that any thread monopolizes a shared resource. An example of this situation occurs when a load misses in

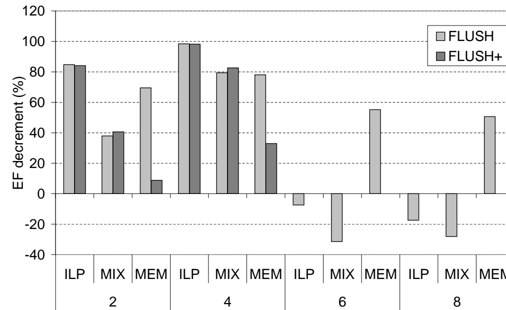


Fig. 3.13: EF decrement of flush++ with respect to flush and flush+

the L2 cache. Current instruction fetch policies focus on this problem and achieve significant performance improvements over icount.

We have compared three different policies addressing this problem. We show that none of the presented policies clearly outperforms all others for all metrics. Results vary depending on the particular workload and the particular metric (throughput, fairness, energy consumption, etc).

The main contributions of this section are two. First, we analyze the use of a load miss predictor in a SMT processor. And second, we present improved versions of the three best policies addressing the described problem that have been published. Our results show that these enhanced versions achieve a significant improvements over the original ones:

- Throughput results indicate that l2mp+ outperforms l2mp for MIX workload by 16% on average and is worse than l2mp only for 2-, 4- and 6-MEM workloads (8% on average). Hmean results show that l2mp+ outperforms l2mp especially for 2-thread workloads.

- The flush+ policy outperforms flush in both throughput and fairness, especially for MEM workloads. Furthermore, it reduces EF by 60% for MEM workloads and only increases EF by 20% for MIX workloads.

- Throughput results show that in general stall+ improves stall, and only for the 2-MEM workload there is a remarkable slowdown of 3%. The Hmean results show that stall+ outperforms stall for all workloads, and especially for MEM workloads.

- Flush++, a new dynamic control mechanism, is presented. It fully outperforms flush policy in both throughput and fairness. The flush++ policy also reduces EF for the 2- and 4-thread workloads, and moderately increases EF for the 6-MIX and 8-MIX workloads. Flush++ outperforms the stall+ policy, with just a small degradation in throughput in the 6-MIX workload.

3.5 DWarn

As shown in the previous section, an inappropriate use of the shared resources by a thread can seriously damage the performance of the remaining threads. If the processor would know in advance which loads are going to miss in the L2 cache this problem would be alleviated. A clear in-advance indicator of L2 misses are L1 misses, because a load only misses in L2 if it previously missed in L1. Data Gating policy [20] is based on this fact, and stalls a thread every time it has an L1 miss. However, not all L1 misses cause an L2 miss. Our results show 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 is too severe and causes resource under-use. On the other hand, to react when the L2 miss is declared only alleviates the problem, because the problem still persists until then, what may be too late. The flush policy [63] works in this way and when the L2 miss is detected the thread it belongs to is flushed from the pipeline and fetch-stalled until the offending load is resolved. However, our results show that for memory bounded threads, the flushed instructions represent 35% of all fetched instructions. Obviously, this increases power consumption since many instructions need to be fetched several times.

The policy we propose in this section, dwarn, also addresses this problem. The implementation of dwarn implies neither extensive hardware complexity nor additional power consumption. The dwarn policy does not squash instructions in the pipeline. Furthermore, it adapts to pressure on resources reducing resource under-use. Dwarn uses L1 data cache misses as indicators of a possible L2 miss. Threads experiencing an L1 data cache miss are given lower fetch priority than threads with no data cache misses. The key idea is to prevent the damage before it occurs, instead of waiting until an L2 miss is produced, when probably some damage has already been done. However, given that we are preventing possible damage (an L2 miss) and we are not sure that this damage will really occur (not all L1 misses cause an L2 miss), we prefer to lower the priority of threads instead of a more drastic measure like stalling threads entirely.

3.5.1 Simulation methodology

Our baseline configuration is shown in Table 3.4, which represents a 9-stage-deep pipeline. Two important issues related to fetch policies are the following. First, the fetch is aware that a load has missed in the L1 cache 5 cycles after this load is fetched into the processor (if no resource conflicts happen). Second, it takes 10 cycles more from the L1 data miss to access the L2 cache (again, if no resource conflicts happen).

In [63] stall and flush are evaluated for 2-, and 4-thread workloads. In [20] dg and pdg are evaluated only for 8-thread workloads. Properties of workloads vary depending on the number of threads they have. In addition, their properties also depend on the cache behavior of the threads. In order to make a broad and fair comparison of the policies, and to avoid that our results are tuned for a special

Table 3.4: Baseline configuration

Processor Configuration	
Fetch/Issue/Commit Width	8
Baseline fetch policy	icount
Fetch mechanism	2.8
Issue Queue Entries	32 int, 32 fp, 32 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	384 int, 384 fp
ROB size	256 entries per thread
Branch Prediction Configuration	
Branch Predictor	2K 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, 2-way, 8-bank, 64-byte lines, 10 cycle access
Main memory latency	100 cycles
TLB miss penalty	160 cycles

Table 3.5: Workload classification based on cache behavior of threads.

# of threads	Workload type	Workloads
2	ILP	gzip, bzip2
	MIX	gzip, twolf
	MEM	mcf, twolf
4	ILP	gzip, bzip2, eon, gcc
	MIX	gzip, twolf , bzip2, mcf
	MEM	mcf, twolf, vpr, **twolf**
6	ILP	gzip, bzip2, eon, gcc crafty, perlbmk
	MIX	gzip, twolf, bzip2, mcf, vpr, eon
	MEM	mcf, twolf, vpr, parser, **mcf**, **twolf**
8	ILP	gzip, bzip2, eon, gcc crafty, perlbmk, gap, vortex
	MIX	gzip, twolf, bzip2, mcf, vpr, eon, parser, gap
	MEM	mcf, twolf, vpr, parser, **mcf**, **twolf**, **vpr**, **parser**

workload, we have created 12 workloads, as shown in Table 3.5. These workloads range from 2 to 8 threads. In the ILP workloads all benchmarks have good cache behavior. All benchmarks in the MEM workloads have an L2 miss rate higher than 1%. Finally, the MIX workloads include ILP threads as well as MEM threads. For MEM workloads some benchmarks are used twice, because there are not enough SPECINT benchmarks with bad cache behavior. The replicated benchmarks are set in boldface in Table 3.5. We have shifted second instances of replicated benchmarks by one million instructions to avoid that both threads access the cache hierarchy at the same time.

3.5.2 The Dcache Warn policy (DWarn)

Our dwarn policy is designed to prevent the negative effects caused by loads that miss in the L2 cache and to reduce the resource under-use. Dwarn uses the $l1$ Detection Moment (DM) and defines a new Response Action (RA), namely, *reduce priority*. There are two main reasons why we use the $l1$ DM. First, unlike the $fetch$ DM, the $l1$ DM detects all loads that miss in L2, because a load misses in the L2 cache if and only if it has previously missed in the L1 cache. Second, to wait X cycles after the issue of the load to take measures for the delinquent thread may be too late, because during that time the delinquent thread can have allocated many resources. In conclusion, the $l1$ DM is both reliable and early enough, avoiding that resources are clogged before the RA is carried out.

The novel *reduce priority* RA is based on the combination of two ideas, namely, classification of threads, and prioritization of threads instead of gating threads.

- Classification: at each cycle available threads are classified into two groups: the first group, called Dmiss group, contains those threads that have one or more in-flight L1 data cache misses. The remaining threads belong to the second group, called Normal group. Once this dynamic classification is done, we know which threads are less-promising (Dmiss) to fetch from. We consider them to be less-promising because instructions after a missing load are more likely to be in the processor for a longer time than those instructions belonging to a thread with no in-flight data cache misses.

- Prioritization: once the classification is done, the fetch priority of the less-promising (Dmiss) threads is reduced. This is done by prioritizing fetch from Normal threads, and fetching instructions from the Dmiss threads only if there are not enough available instructions from the Normal threads. This situation happens, for example, when all Normal threads experience an instruction cache miss, or when there is only one Normal thread available. Threads in each group are sorted using icount.

By reducing the priority of the Dmiss threads, the opportunity of keeping the fetch bandwidth fully used is given to Normal (more-promising) threads. Threads are never stalled, and hence, even if a thread is in the Dmiss group, it has some opportunity to fetch instructions. The idea behind this is that not all L1 misses cause an L2 miss. The fourth column of Table 3.6 shows the percentage of L1 data misses that cause an L2 miss. For MEM workloads less than 50% of L1 misses cause an L2 miss (except for the mcf benchmark). Hence, to stall a thread on each L1 data miss would be too strict a measure.

2-thread workloads

When 2-thread workloads are executed, especially the 2-MIX and the 2-MEM workloads, very specific problems arise, which the *reduce priority* RA cannot prevent.

Table 3.6: Cache behavior of each benchmark

	L1 miss rate	L2 miss rate	L1 → L2
mcf	32.3	29.6	91.6
twolf	5.8	2.9	49.3
vpr	4.3	1.9	44.7
parser	2.9	1.0	36.0
gap	0.7	0.7	94.0
vortex	1.0	0.3	33.3
gcc	0.4	0.3	82.2
perlbmk	0.3	0.1	42.7
bzip2	0.1	0.1	97.9
crafty	0.8	0.1	6.9
gzip	2.5	0.1	2.0
eon	0.1	0.0	2.1

When dwarn policy is used in a processor executing a 2-thread workload, both threads can belong either to a different group, or to the same group (Dmiss or Normal).

- If each one belongs to a different group, the classification made by dwarn is effective because we are identifying which is the less-promising thread (the one in the Dmiss group), and thus, the less-promising instructions. Even so, we will explain why the reduce priority RA is not effective in this case.
- If both threads belong to the same group, the classification is not effective, and the actual policy driving the fetch is icount (recall that threads in the same group are sorted using icount). If both threads belong to the Normal group that means that both threads are in an ILP code segment. Thus, there is no performance degradation because icount presents good results when classifying threads with high ILP. Instead, if both threads belong to the Dmiss group performance is heavily degraded as we will see later in this section.

Our result show that for the 2-MIX workload, during 23% of the execution cycles we are in the situation where each thread belongs to a different group. The fetch fragmentation due to branches and cache line boundaries reduces fetch efficiency [11], and hence the Normal (high-priority) thread is not able to fully use the fetch bandwidth. Consequently, given that we fetch instructions for 2 threads each cycle, the Dmiss (less-promising) thread can still fetch instructions into the processor. As a result, many not promising instructions from the Dmiss thread are fetched into the processor, and little by little internal resources are frozen by these instructions, which will remain for a long time in the processor degrading performance.

When the 2-MEM workload is executed, our results show that during 51% of the execution cycles all the threads making fetch are in the Dmiss group. Hence, all the fetched instructions belong to Dmiss threads, and icount is the policy driving the fetch. In this situation, as we will see in Figure 3.15, the allocation of resources is poor and they are frozen for a long time. In addition, 20% of the execution cycles

each thread belongs to a different group, so the problem presented for the 2-MIX workload also persists in this case.

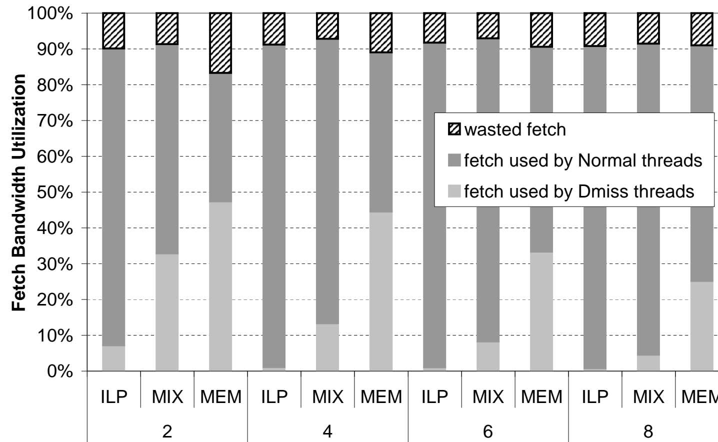


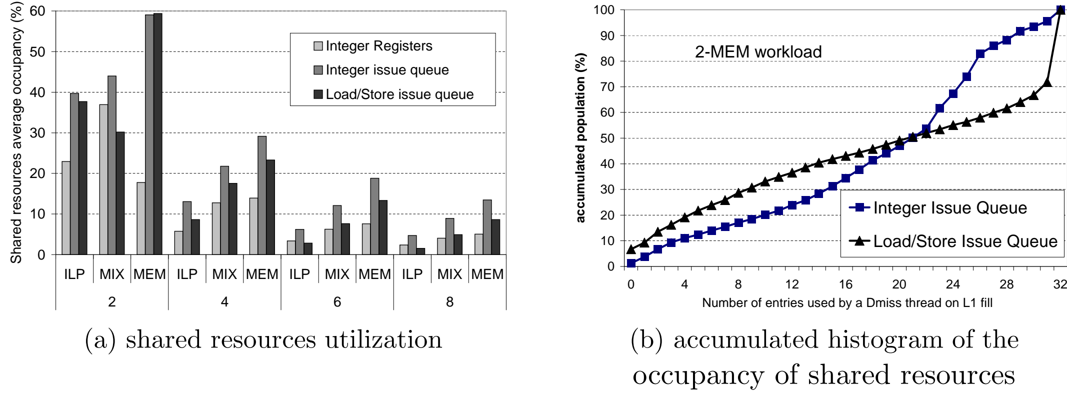
Fig. 3.14: dwarn fetch bandwidth utilization

Figure 3.14 quantifies the use of the fetch bandwidth made by each group of threads. Bars respectively indicate the percentage of fetch bandwidth used by Dmiss threads, Normal threads, and the wasted percentage. For the 2-MIX workload 33% of the fetch bandwidth is used by Dmiss threads. This value is even greater for the 2-MEM, where 47% of the fetch bandwidth is used by Dmiss threads. Consequently, many instructions that make little progress and use shared resources for a long time are still fetched into the processor, degrading performance.

Figure 3.15(a) shows the percentage of the shared resources used by a thread having an L2 miss when this load is finally serviced when the reduce priority RA is used. Bars respectively show the average occupancy of the integer registers (IREGS), the integer issue queue (IQ), and the load/store issue queue (LQ). For 2-thread workloads average values are 30% of the IREGS, 50% of the IQ and 40% of the LQ. These values gradually decrease as the number of threads increases, because there are more threads competing for resources. For the 2-MEM workload almost 60% of the IQ and the LQ are frozen by Dmiss threads. Although on average 40% of these queues are available for the other thread, we see in the accumulated frequency histogram (Figure 3.15(b)) that 30% of the loads missing in L2 use more than 24 entries of the IQ (75%) and the whole LQ. That is, many times there are few available entries in the queues, degrading performance of the other threads even when we use dwarn.

Summary

Ultimately, to harness an 8-instruction-wide processor we need either to fetch instructions from several threads in a single cycle, or use new fetch organizations that provide high fetch performance for 1.X instruction fetch policies [23]. In the former



(a) shared resources utilization

(b) accumulated histogram of the occupancy of shared resources

Fig. 3.15: Dwarn resource utilization

case, when we use a 2.8 fetch mechanism and only 2 threads are executed, the issue queue entries and physical registers can be occupied for a long time by threads that make little or no progress. The main reason is that the classification made by dwarn is not effective because even when the Dmiss threads are given low priority, the processor often fetches instructions from them[1].

Different workloads present different behavior depending on the number of threads and their cache miss rate. Knowing the behavior of a thread at runtime is not easy, but the number of running threads is known by the processor at any time. Hence, a way of addressing this variable behavior consists of using an I-Fetch policy that varies its behavior based on the number of executing threads. We propose a hybrid mechanism that uses different RAs based on this number. If there are less than three threads running we will use two RAs. When a load misses in L1 the priority of its (Dmiss) thread is reduced. After that, if the load finally misses in L2 its thread is gated. In this way we prevent the Dmiss thread from clogging shared resources. If the number of running threads is higher than 2 we will just reduce the priority of Dmiss threads, because this is enough to prevent Dmiss threads from clogging resources.

Implementation

Additional hardware required to implement our technique consists of a data miss counter for each hardware context. These counters are incremented every time a thread experiences a data cache miss and decremented when the data cache fill occurs. If the counter of a thread is zero this thread will belong to the Normal group, otherwise it will belong to the Dmiss one.

[1]In Section 3.5.4 dwarn is evaluated for an 1.X fetch mechanism

3.5.3 Performance evaluation

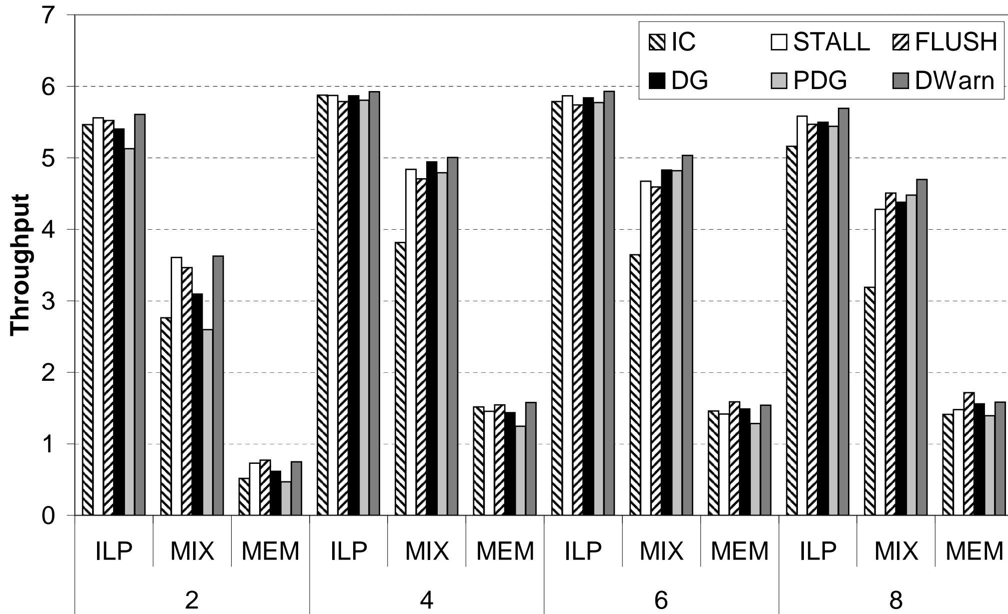
In this section we compare the efficiency of dwarn with the flush [63], stall [63], dg [20], and pdg [20] policies. Before showing our results, we discuss several important issues about the implementation of the stall, flush, dg and pdg policies. Concerning stall and flush, in [63] a load is supposed to miss in L2 when it spends more than 15 cycles in the memory hierarchy. We have experimented with different values for this parameter and we found that 15 presents the best overall results for our baseline architecture. Stall and flush have the following three additional properties: first, a data TLB miss also triggers a stall (flush); second, a 2-cycle advance indication is received when a load returns from memory; and third, this mechanism always keeps one thread running. That is, if there is only one thread running, it is not stopped even when it experiences an L2 miss.

About dg, we have experimented with the number of L1 missing loads that a thread can experience before it is stalled: a low value can lead to over-stalling, a high value causes that when there are few L1 misses these are not filtered. That is, the thread they belong to is not stalled causing internal shared resources to be clogged. The value $n = 0$, the same used in [20], presents the best overall results.

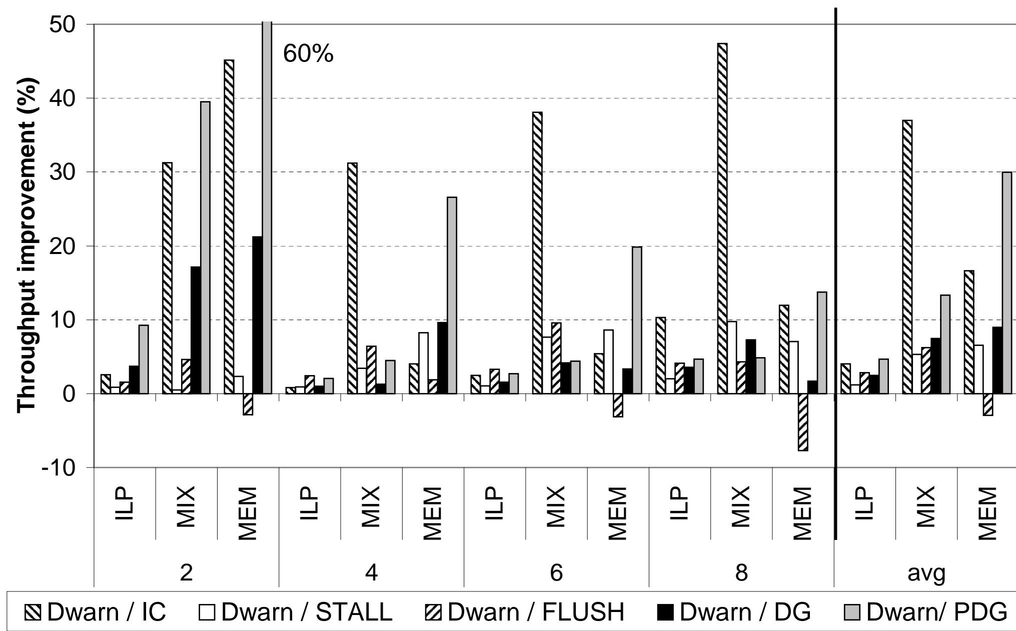
Throughput results

Figure 3.16(a) shows the absolute IPC values for the different policies and Figure 3.16(b) shows the throughput improvement of dwarn over each of the others policies. It shows that dwarn outperforms all previous policies, except flush for the MEM workloads. On average, dwarn outperforms icount by 18% and this improvement is higher as the number of threads increases: when there are many threads running, the pressure on shared resources is very high. Consequently, tolerating the latency of L1 misses causes a significant performance drop in the other threads. Dwarn reduces the fetch priority of threads experiencing L1 misses alleviating this problem. Icount does not act on an L1 miss and suffers a significant performance penalty.

Regarding dg, dwarn outperforms it for all workloads: 3% for the ILP, 8% for the MIX and 9% for the MEM workloads. This improvement gradually decreases as the number of threads increases. This is because as the number of threads increases, competition for resources also increases and then it is more difficult for a thread to allocate resources. The dg policy gates a thread on each L1 miss, so it sacrifices MEM threads in order to give more resources to ILP threads. However, if the number of threads is low, there are not enough ILP threads to use all these available resources. This means that MEM threads are unnecessarily stopped. Even worse, for MEM threads, less than 50% of L1 misses cause an L2 miss (recall the fourth column of the Table 3.6(a)), hence the number of unnecessary stalls is higher. With the dwarn policy no thread is stopped (for 4 or more threads), but a thread experiencing an L2 miss is executed at a lower priority. In this way, if none of the available Normal threads with higher priority can use the resources they are given to this thread. Thus, resource-underutilization is reduced.



(a) Throughput results of the policies



(b) Throughput improvement of dwarn over the other policies

Fig. 3.16: Throughput results

Regarding pdg, it suffers the same problems as dg, but, in addition, pdg presents 2 additional problems: the first is the predictor mistakes. If a predictor erroneously indicates that a load will miss in the L1 cache and in fact the load does not, this causes an additional unnecessary stop, degrading the performance of the thread. The

second problem is the load serialization. Our results indicate that many cache misses occur close in time. Hence, to stop on each missing load in the fetch stage causes load serialization and loss of the available memory parallelism. Dwarn improves pdg by 5% for the ILP, 13% for the MIX and 30% for the MEM workloads.

Regarding stall, dwarn improves it for all workloads. The improvements are of 2%, 6% and 7% for the ILP, MIX, and MEM workloads respectively.

Dwarn also outperforms flush, for the ILP and MIX workloads, by 3% and 6% respectively, and only suffers a slowdown of 3% for the MEM workloads. The main cause is that for the 6-MEM and 8-MEM workloads the pressure over resources is too high, hence it is more preferable to free resources by flushing the delinquent threads than to freeze resources by stalling these threads. However, this improvement of 3% achieved by flush comes at a high cost. First, dwarn does not require such a complex hardware as flush, and second, it does not re-execute instructions. As shown in Figure 3.17, the number of squashed instructions due to the flush policy represents a significant percentage of the fetched instructions (35% for the MEM workloads).

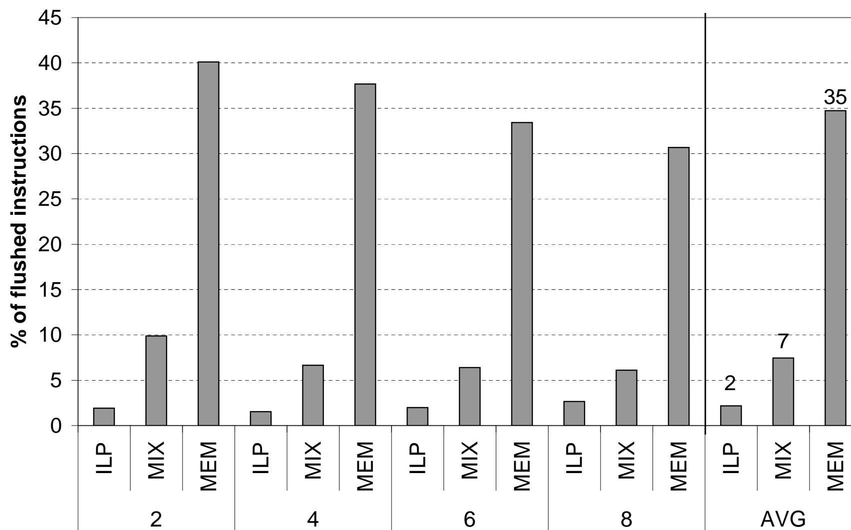


Fig. 3.17: Extra Fetch caused by the flush policy

The problem of flush is that, like stall, it reacts too late, namely, when the L2 miss has been detected, and in a drastic way by flushing all the instructions after the load. The former problem implies that as long as the L2 miss is not declared, instructions after the missing load compete for resources with instructions of other threads that could contribute to final IPC, degrading performance. The latter problem directly affects the thread being flushed, reducing its performance. Dwarn acts earlier, when an L1 miss occurs, and in a controlled manner by reducing the presumed delinquent thread's priority. As a result, only when the remaining threads are not able of using machine resources, these are given to the delinquent thread and hence its performance is less affected.

Table 3.7: Relative IPC of each thread in the 4-MIX workload

Policy	Relative IPCs thread 1 (ILP)	thread 2 (ILP)	thread 3 (MEM)	thread 4 (MEM)	Hmean
icount	0.36	0.41	0.50	0.79	0.47
stall	0.42	0.65	0.38	o.63	o.49
flush	0.41	0.64	o.34	0.59	0.46
dg	0.43	0.70	0.34	0.46	0.45
pdg	0.40	0.72	0.28	0.31	0.38
dwarn	0.44	0.69	0.43	0.70	0.53

Harmonic Mean results

The second metric we have used is the Harmonic Mean (Hmean) [46]. The Hmean metric attempts to avoid artificial improvements achieved by giving more resources to threads with high ILP.

Figure 3.18 depicts the Hmean improvement of dwarn policy over the other policies. Average results indicate that dwarn improves all other policies for ILP, MIX and MEM workloads, only suffering a slowdown of 2% respect to flush. Let us illustrate why dwarn outperforms all previous policies by an example. Table 3.7 shows the relative IPC of each thread in the 4-MIX workload for all policies. The second and the third columns indicate the relative IPC of the two ILP threads in the workload. Likewise, the two following columns indicate the relative IPC of the two MEM threads. The point is that dwarn achieves an IPC for the ILP threads that is as high as the one obtained by the other policies, but it does not significantly affect the IPC of the MEM threads as the other policies do. The reason for this is that dwarn never stalls or squashes threads even when they are in the Dmiss group for four or more threads. As a result, if there are available resources and the Normal threads cannot use them, these resources are given to the Dmiss threads. Regarding icount, we observe that icount achieves the highest result for the MEM threads, but it heavily degrades the performance of the ILP threads.

Concluding, the dwarn policy presents the best balance between achieving high IPC for the ILP threads and harming as little as possible the IPC of MEM threads. As a result, on average, dwarn achieves better Hmean results than the other policies for all types of workloads. It only experiences a slowdown of 2% with respect to flush for the MEM workloads. However, as we saw in Figure 3.17, flush achieves this is at the cost of increasing the number of fetched instruction by 35%.

We have seen that depending on the number of running threads, workloads present very different properties. When there are few threads, reducing priority is not enough to avoid Dmiss threads from using shared resources for a long time. This is because thread level parallelism is low and Normal threads do not use all the fetch

bandwidth. When there are 6 and 8 threads, the pressure on resources is very high and hence the competition for them is high too. A general purpose I-fetch policy must be aware of the number of executing threads to better adapt to the properties of each workload. Overall results show that our hybrid mechanism, which combines the *gate* and the *reduce priority* RAs when there are two threads running, and uses the *reduce priority* RA in the remaining cases, outperforms all previous proposed instruction fetch policies.

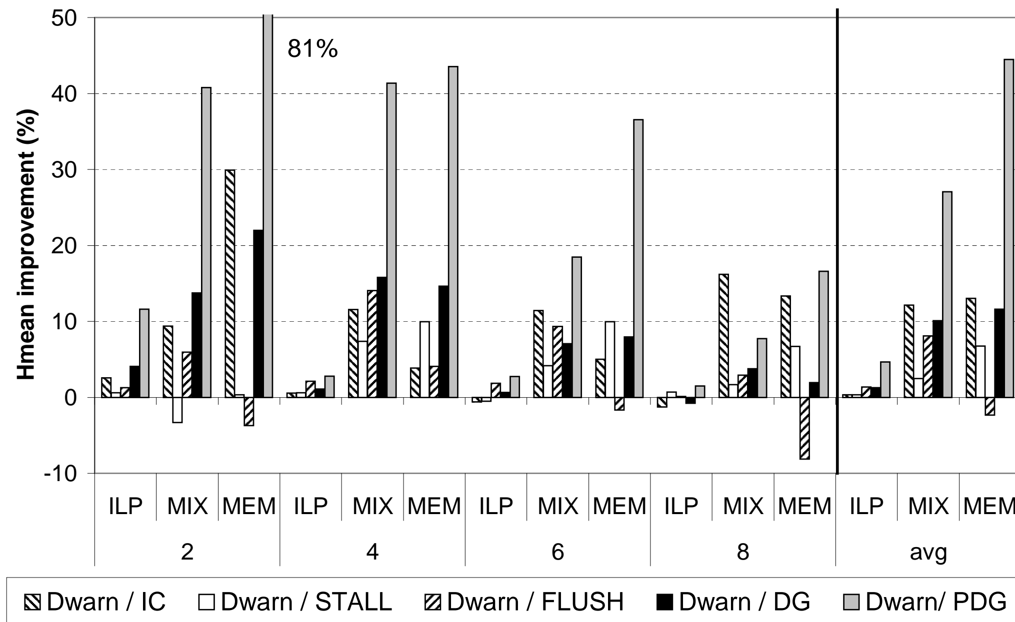
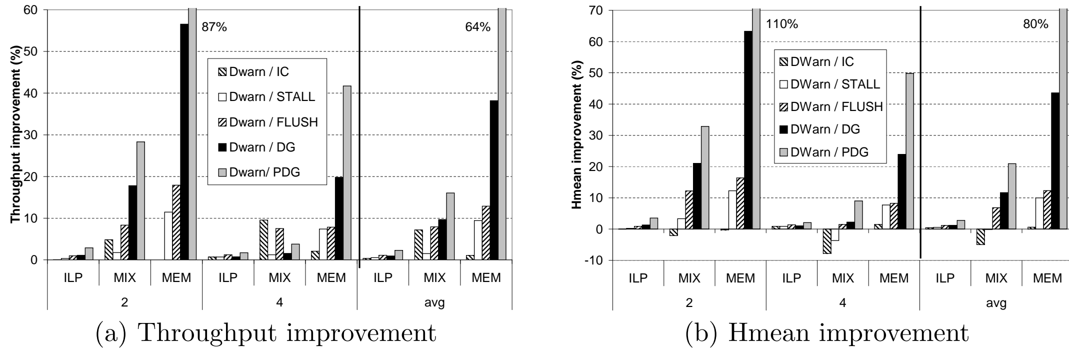


Fig. 3.18: Hmean improvement of dwarn over the other policies

3.5.4 DWarn on different architectures

The time needed to determine whether a load has caused an L1 data cache miss or an L2 miss are two key factors in the previous policies. Another important factor is the number of threads that can fetch instructions into the processor each cycle. In this section, we experiment with two variants of the original architecture in order to analyze the effect of these factors.

The first architecture represents a less aggressive processor than presented in Table 3.4. This is a 4-wide, 4-context processor with an 1.4 fetch mechanism. There are 256 physical registers and 3 integer, 2 floating point, and 2 load/store functional units. In this architecture, we can fetch instructions only from one thread each cycle, and hence, the Dmiss threads cannot fetch if there is at least one Normal thread. On the one hand, it is beneficial because Dmiss threads can unlikely clog resources. On the other hand, MEM threads are now more damaged. Figures 3.19 (a) and (b) show the throughput and Hmean improvement of the dwarn policy over the other ones.



(a) Throughput improvement (b) Hmean improvement

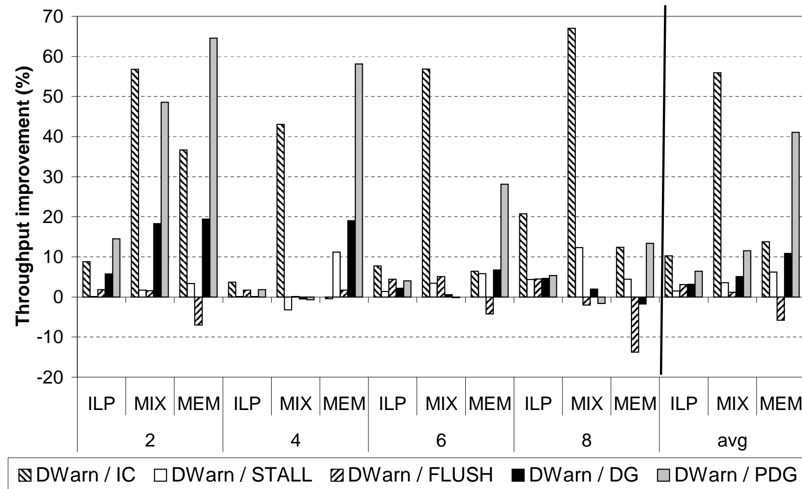
Fig. 3.19: Dwarn improvement over the other policies (small architecture)

Hmean results show that for the MIX workloads dwarn is outperformed by icount, by 5% on average. The main cause for this slowdown is that the MEM threads are now heavily damaged. Regarding the other policies addressing the load miss latency problem, dwarn clearly outperforms them in both throughput and Hmean. The throughput improvements for the MIX and the MEM workloads are: 5% over stall, 23% over dg, 10% over flush, and 40% over pdg. Hmean improvements for the MIX and the MEM workloads are: 5% over stall, 28% over dg, 10% over flush, and 50% over pdg.

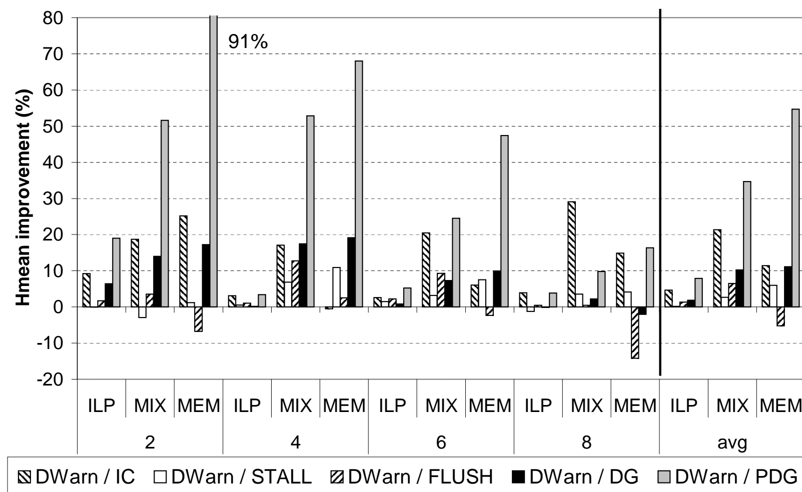
The second architecture represents a deeper and more aggressive processor than presented in Table 3.4. This is a 16-stage-depth processor, with a 2.8 fetch mechanism, and 64-entry issue queues. The time to determine an L1 miss has been incremented by 3 cycles, the latency from the L1 cache to the L2 from 10 to 15 cycles, and the memory latency has also been incremented to 200 cycles. Figures 3.20 (a) and (b) show the throughput and Hmean improvement of the dwarn policy over the other ones. As we see in the average results, dwarn throughput and Hmean results indicate that it improves all other policies for all type of workloads, except for the MEM where it suffers a slowdown of 6% with respect to flush. The main cause for this high average slowdown is the 8-MEM workload. In that case, there is an over-pressure on resources (our results show that the throughput for the 4-MEM workload is almost the same than for the 6-, and the 8-MEM workloads), whereby flushing is much more effective than stalling threads. However, our results (not shown here) show that this is at the cost of increasing the number of fetched instructions due to flushes. This increment is 56% on average for the MEM workloads.

3.5.5 Conclusions

The performance of an SMT processor directly depends on how the dynamic allocation of shared resources is done. The instruction fetch policy dynamically determines how this allocation is carried out. To achieve high performance, the fetch policy



(a) Throughput improvement



(b) Hmean improvement

Fig. 3.20: Dwarn improvement over the other policies (deep architecture)

must avoid the monopolization of a shared resource by any thread. An example of this situation occurs when a load misses in the L2 cache level. Icount [64] reacts too late and suffers significant performance degradations, in particular for 2-thread workloads where this problem is most acute. In this case, flush [63] clearly outperforms icount. However, the problem of flush is that it does not prevent damage, but drastically cures it once it has happened. Other policies [20] try to prevent this damage by acting before the L2 miss occurs. Dg stalls threads on an L1 miss and pdg on a predicted L1 miss in the fetch stage. However, when there are few threads, these policies are too strict and cause resource under-utilization and an important performance degradation.

In this subsection, we have proposed a novel policy that deals with this problem (dwarn). On average, dwarn throughput results show that it improves all other policies for all types of workloads, especially for the MIX and MEM ones: 27% over icount, 6% over stall, 2% over flush, 8% over dg and 22% over pdg. Dwarn only suffers a loss of 3% with respect to flush for the MEM workloads. However, this comes at the cost of increasing hardware complexity for these policies and the number of fetched instructions (35% for memory bounded threads). Fairness results show that dwarn clearly presents the best throughput-fairness balance, only suffering a slowdown of 2% with respect to flush for the MEM workloads. The improvement for the MIX and MEM workloads are: 13% over icount, 5% over stall, 3% over flush, 11% over dg and 36%over pdg. If we take into consideration all these results, dwarn presents as a good solution to the problem of long latency loads for throughput, fairness, complexity, and power.

3.6 Chapter summary

In this chapter we have presented two techniques with the objective of improving the throughput and fairness in high-performance systems. We have compare our proposals with the best instruction fetch policies published: flush, stall, data gating, predictive data gating and l2mp.

- Flush++ adapts its behavior to the dynamic number of live “threads” available to the fetch logic. Due to this additional level of adaptability, on average flush++ improve all previous policies in throughput and fairness while achieves a reduced extra fetch.
- Dwarn is not predictive and requires minimum hardware resources. It does not flush instructions reducing overall processor complexity and wasted power. Furthermore, dwarn adapts to pressure on resources better than the other policies. If there are few running threads, it avoids resource under-utilization. When the number of threads increases, reducing fetch priority is enough to avoid Dmiss threads from holding resources for a long time.

The development of these two policies as well as the study of the previous fetch policies were the first step to envisage the concept of *explicit resource allocation*, which is addressed in more detail in the next chapter.

4

From Implicit Resource Allocation to Explicit Resource Allocation

CHAPTER 4

FROM IMPLICIT RESOURCE ALLOCATION TO EXPLICIT RESOURCE ALLOCATION

In this chapter we explain our concept of *explicit resource allocation*. We start explaining the need of explicitly managing resource allocation in SMT processors. Next, we explain how explicit resource allocation improves throughput/fairness of SMTs. Finally, we explain how the explicit resource allocation enables the concept of *quality of service* in SMTs.

4.1 Introduction

In current SMT processors, threads are executed in a common resource pool and are allowed to freely compete for resources. The instruction 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 of using 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, like data L1 cache misses. We call this resource management *indirect control of shared resources* or *implicit resource allocation*.

The implicit use of resources causes a two-fold problem: throughput/fairness degradation and lack of predictability.

- Throughput and fairness degradation: because no direct control over shared 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, i.e., after a L2 cache miss. There are fetch policies in the literature [20][43][63] 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.

- Lack of predictability: as we mentioned before, it is the instruction fetch policy that decides from which threads instructions are brought into the processor. In this sense, the fetch policy acts like a hardware scheduler that works independently from the software scheduler inside the Operating System. This means that the OS level scheduler can have negative interference with the hardware scheduler inside the processor. Once instructions are inside, they are assigned resources blindly.

Thus, in order to exploit the available resources in an SMT processor maximally, implicit resource allocation by means of the instruction fetch policy is not adequate and a more explicit resource allocation scheme is required. In the next two sections, we analyze how explicit resource allocation could alleviate the degradation of throughput and fairness (section 4.2) and achieve predictability in SMT systems (section 4.3).

4.2 Increasing throughput and fairness

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 other critical shared resources. This direct control allows a better use of resources, reducing resource under-utilization.

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 4.1 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.

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 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.

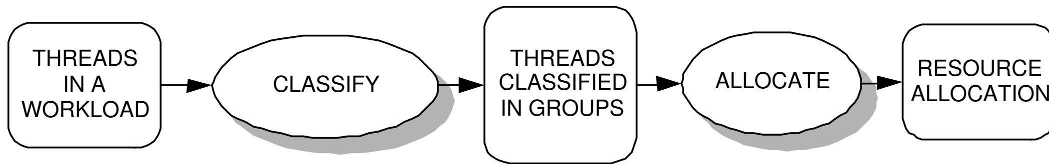


Fig. 4.1: Main tasks of a resource allocation policy

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 study 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.

Resource allocation

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

- 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 section, 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 4.1.

Table 4.1: Resource allocation used

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

4.2.1 Resource allocation policies vs. instruction fetch policies

The main differences between 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.

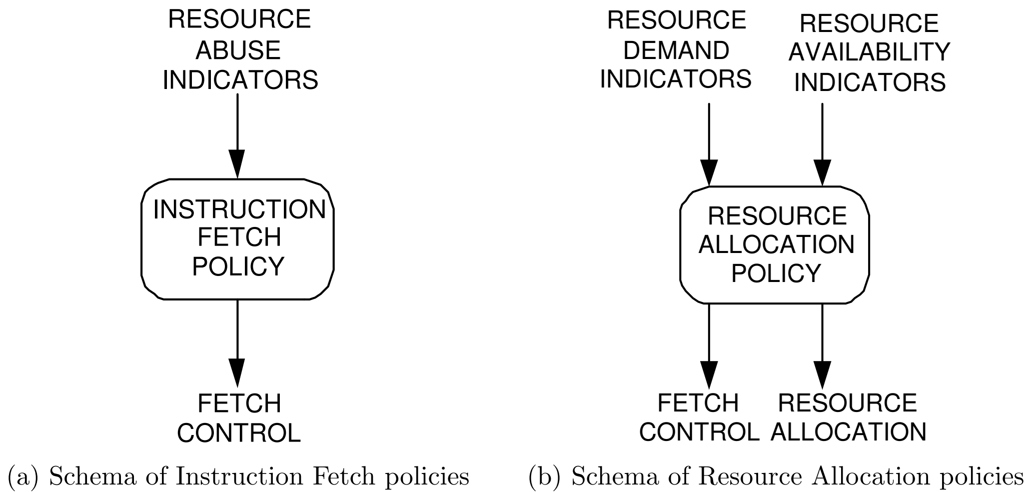
I-fetch policies just directly control the fetch bandwidth. All I-fetch policies we have seen stall the fetch of threads. Flush, in addition, squashes all instructions of the offending thread after a missing load. As input information, I-fetch policies use indirect indicators, like L1 misses or L2 misses. Table 4.2 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.

Table 4.2: Resource actions and input information of fetch policies

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

A RAlloc policy controls fetch bandwidth, as I-fetch policies do. As shown in [64], 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 abuse indicators and, as a consequence, when any of them is activated the I-fetch policy immediately stalls or flushes threads (see Figure 4.2.1 (a)). An RAlloc policy, see Figure 4.2.1 (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



(a) Schema of Instruction Fetch policies (b) Schema of Resource Allocation policies

Fig. 4.2: Schema of I-Fetch and RAlloc policies

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.

4.2.2 Methodology

Table 4.3 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.

Programs are divided into two groups based on their cache behavior as in previous chapters. It is vital to differentiate between 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 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 section, we consider workloads with only 2 threads like the Pentium 4 [47] or the Power 5 [36].

We have used the workloads shown in Table 4.4. 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.

Table 4.3: Baseline configuration

Processor Configuration	
Pipeline depth	12 stages
Fetch/Issue/Commit Width	8
Issue Queue Entries	32 int, 32 fp, 32 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	256 integer, 256 fp
ROB size (each thread)	256
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, 10 cycle access
Main memory latency	100 cycles
TLB miss penalty	160 cycles

Table 4.4: Workload classification based on cache behavior of threads.

	Workload type		
number	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

4.2.3 Performance evaluation

We compare our smartRA policy with some of the best proposed fetch policies currently known: icount [64], stall[63], flush[63], dg[20] and pdg[20]. For clarity, we only show the results of those policies that give better results for the setup examined in this section: flush, icount, and dg.

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.

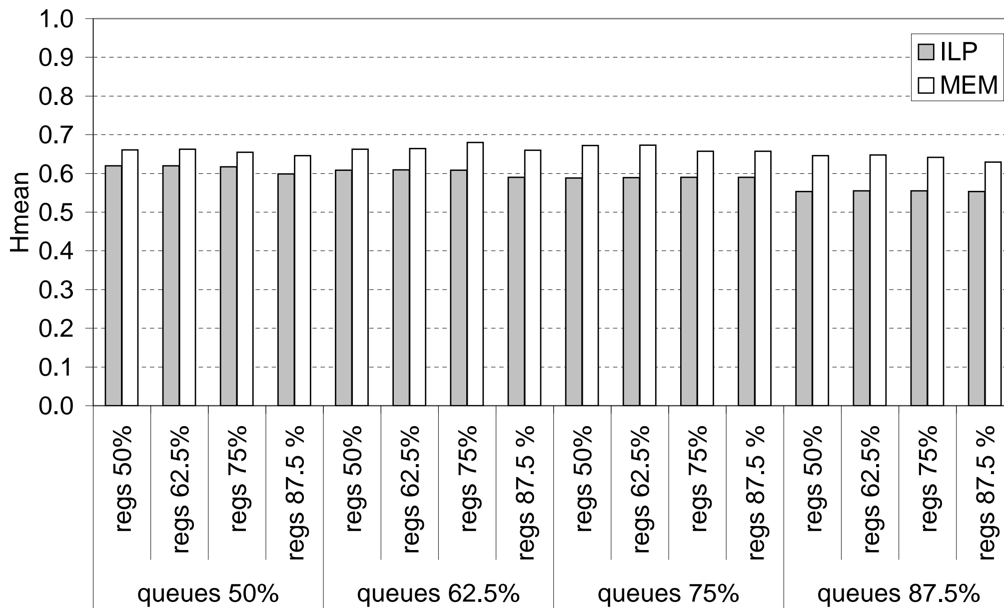
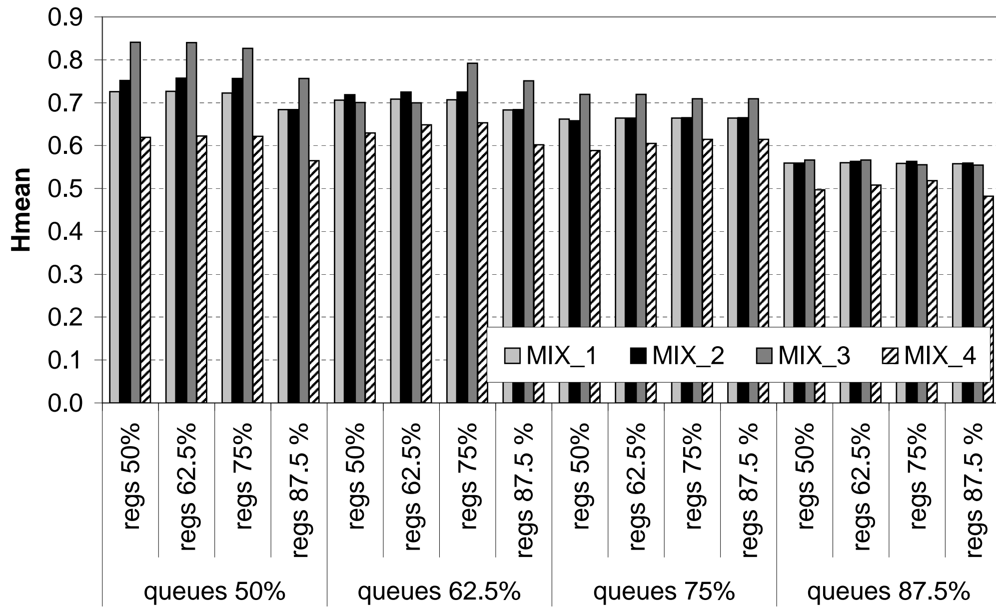


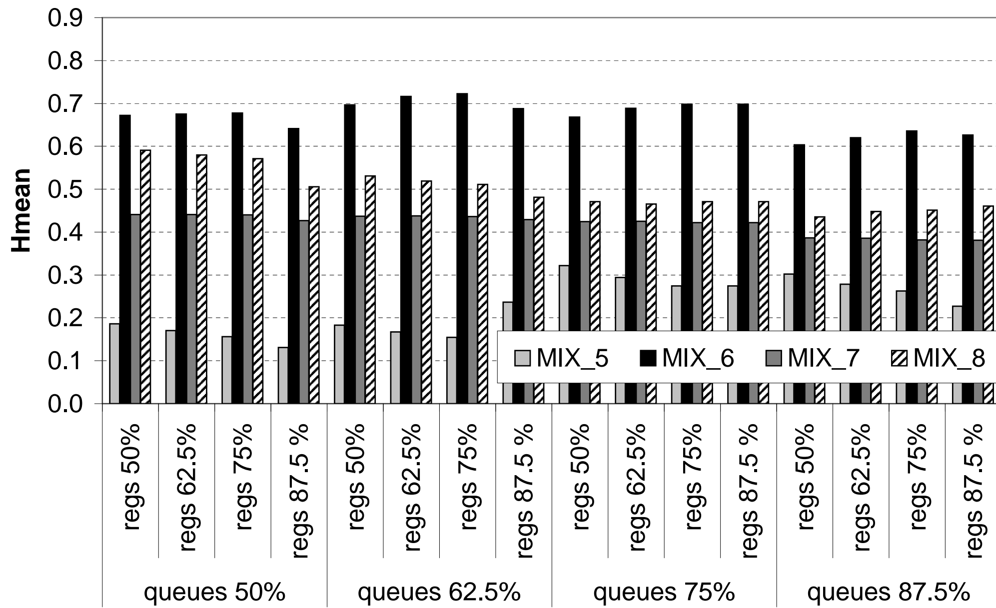
Fig. 4.3: Comparison of the different resource allocations for ILP and MEM workloads

Figure 4.3 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 4.4 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



(a) Workloads from 1 to 4



(b) Workloads from 5 to 8

Fig. 4.4: Hmean results for all resource allocations for all MIX workloads

leads to the best result for all workload types. Instead, each workload achieves the best Hmean result with a different resource allocation. Tables 4.5(a) and 4.5(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. Recall that these divisions are applied when there is one thread in a slow phase and the other is in a fast phase. If both threads are in the same phase resources are evenly split.

Table 4.5: Issue queue and physical register division that lead to the best Hmean results.

	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 type		
	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

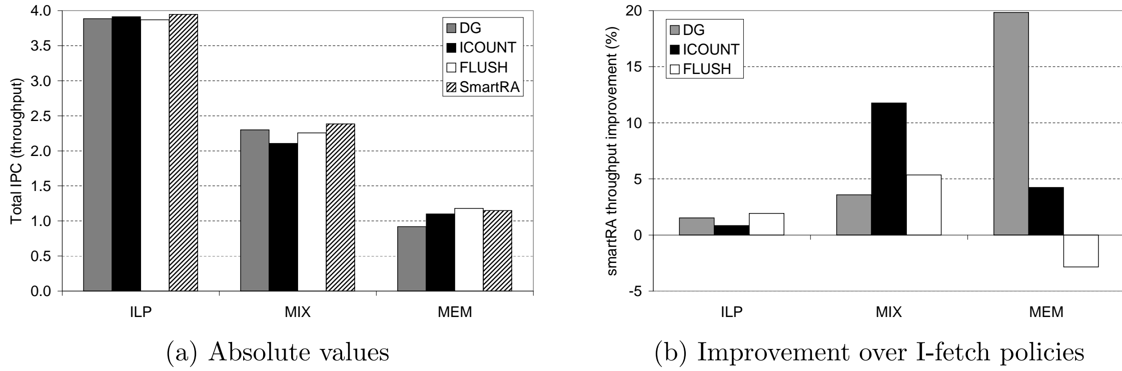
(b) Registers

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.

Resource allocation vs. I-fetch policies

Tables 4.5(a) and 4.5(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 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) Improvement over I-fetch policies

Fig. 4.5: IPC throughput

Hmean results, see Figure 4.6, 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.

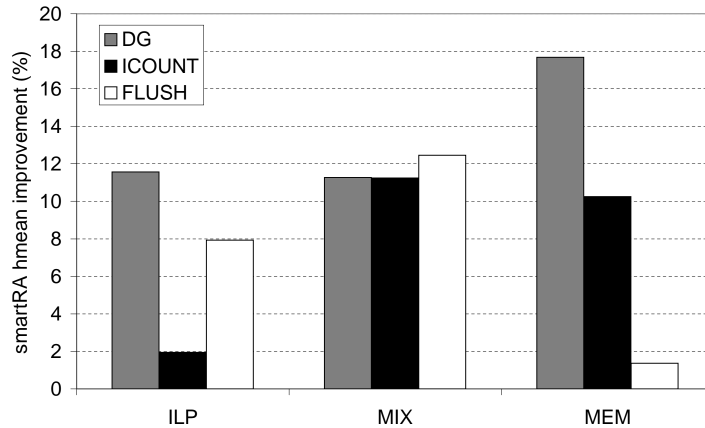


Fig. 4.6: Hmean improvement of smartRA over various I-fetch policies

From the fetch policies, flush achieves the best results. On average, smartRA improves flush by 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 4.7. 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.

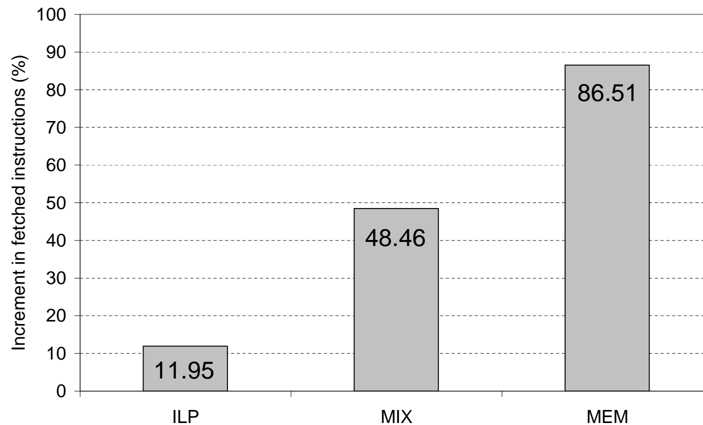


Fig. 4.7: Increment in the number of fetched instructions when the flush policy is used

4.2.4 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.

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.

4.3 Feasibility of QoS in SMT through explicit resource allocation

In an SMT, several threads are running together, sharing resources at the micro-architectural level, in order to increase throughput. A fetch policy decides from which threads instructions are fetched, thereby implicitly determining the way processor resources, like rename registers, are allocated to the threads. However, with current policies the performance of a thread in a workload is unpredictable since it depends on the amount of resources given to that thread. This poses problems for the suitability of SMT processors in the context of (soft) real-time systems.

The key issue is that in the traditional collaboration between OS and SMT, the OS only assembles the workload while it is the processor that decides how to execute this workload, implicitly by means of its fetch policy. Hence, part of the traditional responsibility of the OS has "disappeared" into the processor. One consequence is that the OS may not be able to guarantee time constraints even though the processor has sufficient resources to do so. To deal with this situation, the OS should be able to exercise more control over how threads are executed and how they share the processor's internal resources.

In this section, we discuss our philosophy behind a novel collaboration between OS and SMT in which the SMT processor provides 'levers' through which the OS can fine tune the internal operation of the processor to achieve certain requirements. We want to reserve resources inside the SMT processor in order to guarantee certain requirements for executing a workload. We show the feasibility of this approach by a simple parameterized mechanism that assigns fetch slots and instruction and load queue entries to a High Priority Thread. This, in turn, is a first step toward enabling the OS to execute a thread at a given percentage of its full speed and thus enabling the use of out-of-order, high performance SMT processor in real-time environments.

4.3.1 QoS: A novel collaboration between OS and SMT

We approach OS/SMT collaboration as *Quality of Service (QoS)* management. This approach has been inspired by QoS in networks. In an SMT, resources can be reserved for threads guaranteeing a required performance. We observe that on an SMT processor, each thread reaches a certain percentage of the speed it would achieve when running alone on the machine. Hence, for a given workload consisting of N applications and a given instruction fetch policy, these fractions give rise to a point in an N-dimensional space, called the *QoS space*. For example, Figure 4.8(a) shows the QoS space for two threads, `eon` and `twolf`, as could be obtained for the Pentium4 or the Power5. In this figure, both x- and y-axis span from 0 to 100%. We have used two fetch policies: *icount* [64] and *flush++* [13]. Theoretically it is possible to reach any point in the shaded area below these points by judiciously inserting empty fetch cycles. Hence, this shaded area is called the *reachable part* of the space for the given fetch policies. In Figure 4.8(b), the dashed curve indicates points that intuitively could be reached using some fetch and resource allocation policy. Obviously, by assigning all fetch slots and resources to one thread, we reach

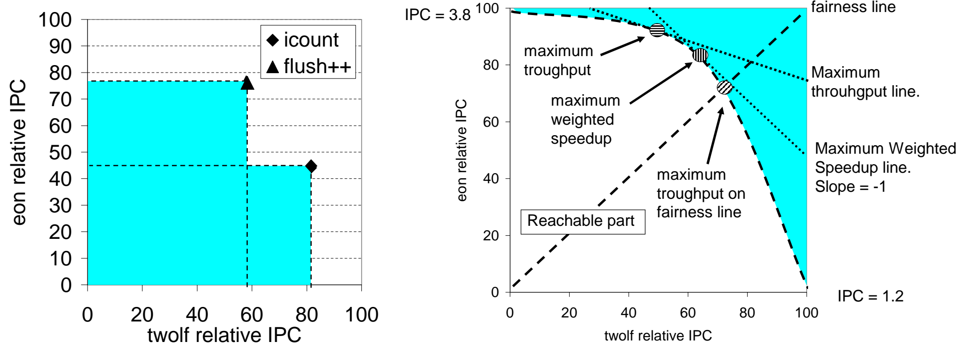


Fig. 4.8: (a) QoS space for three fetch policies; (b) important QoS points and areas

100% of its full speed, that is, the speed it would reach when run alone. Conversely, it is impossible to reach 100% of the speed of each application at the same time since they have to share resources.

Each point or area (set of points) in the reachable subspace entails a number of properties of the execution of the applications: maximum throughput; fairness; real-time constraints; power requirements; a guarantee, say 70%, of the maximum IPC for a given thread; any combination of the above, etc. In other words, each point or area in the space represents a solution to a *QoS requirement*. It is the responsibility of the OS to select a workload and a QoS requirement and it is the responsibility of the processor to provide the levers to enable the OS to pose such requirements. To implement such levers, we add mechanisms to control how these resources are actually shared. These mechanisms include prioritizing instruction fetch for particular threads, reserving parts of the resources like instruction or load/store queue entries, prioritizing issue, etc. The OS, knowing the needs of applications, can exploit these levers to navigate through the QoS space.

In this section, we present a first step toward this goal by studying the behavior of threads when a certain number of resources is reserved for a High Priority Thread (HPT). We show that such a simple mechanism is already capable of influencing the relative speed of threads considerably and hence can cover the QoS space to a great extent.

4.3.2 QoS by resource allocation

We use a standard 4-context SMT configuration. There are 6 integer, 3 FP, and 4 load/store functional units and 32-entry integer, load/store and FP IQs. There are 320 physical registers shared by all threads. Each thread has its own 256-entry reorder buffer. We use a separate 32K, 4-way data and instruction caches and a unified 512KB 8-way L2 cache. The latency from L2 to L1 is 10 cycles, and from memory to L2 100 cycles. We consider 4 workloads in which the High Priority Thread (HPT) is ILP or MB, and the Low Priority Thread (LPT) is ILP or MB. The workloads are: `gzip` and `bzip2` (ILP-ILP), `gzip` and `twolf` (ILP-MB), `twolf` and `bzip2` (MB-ILP), and `twolf` and `vpr` (MB-MB).

Static resource allocation

We statically reserve 0, 4, 8, ..., 32 entries in the IQ and LSQ for the HPT. The remaining entries are devoted to the LPT. Moreover, we prioritize the instruction fetch and issue for the HPT: in each cycle, we first fetch/issue instructions from the HPT. If there are fetch opportunities left, then instructions from the LPT are fetched/issued. There are more resources in an SMT processor that are shared between threads, most notably the rename registers and the L1 and L2 caches. We have restricted attention to IQ and LSQ entries because these shared resources most directly determine which instructions from which thread are executed.

Results

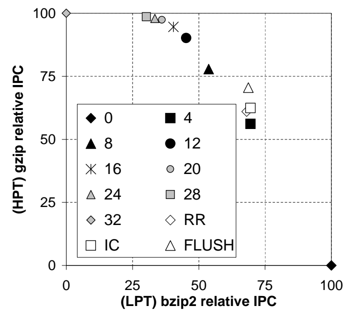
We show the resulting QoS space for varying numbers of assigned resources in Figures 4.9(a) through 4.9(h). We also show the points obtained from the *round robin* (RR), *icount* (IC), and *flush* fetch policies for comparison. We immediately observe that our parameterized mechanism is capable of covering a large part of the reachable space by tuning its parameter. In contrast, the points reached by the three standard fetch policies show no coherent picture. For the ILP-ILP and MB-MB workloads, they reach points that are quite close together. For the other workloads, there is considerable difference and their relative position in the space changes. This shows that standard fetch policies provide little control over the execution of threads.

Figure 4.9 shows the resulting IPC values as well. In the figures showing IPC we also show IPC obtained from the standard policies *icount* and *flush* for comparison.

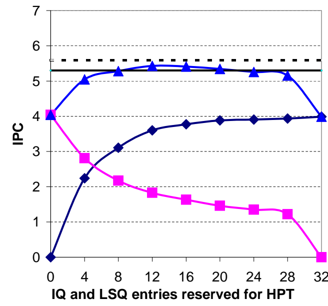
- ILP-ILP: both threads have a high throughput and do not occupy IQs for a long time. As a result, reserving a number of these entries for the HPT and moreover prioritizing its fetch, quickly produces a situation in which the HPT dominates the processor and its speed comes close to its full speed. Entries left, does not completely collapse. The total throughput is about the same as for *icount* (except the cases of 0 and 32) which means that what we take from one thread can successfully be used by the other.

- ILP-MB: when the LPT thread misses in the L2, it tends to occupy resources for a long time which has an adverse effect on the speed of the other thread. Therefore, reserving resources for the HPT that is ILP causes its speed to sharply increase. As a result, the total throughput can be larger than for *icount*. *Flush* needs to re-fetch and re-issue all flushed instructions, degrading its performance. The speed of the LPT does not degrade fast since it suffers many L2 misses and thus cannot use many resources.

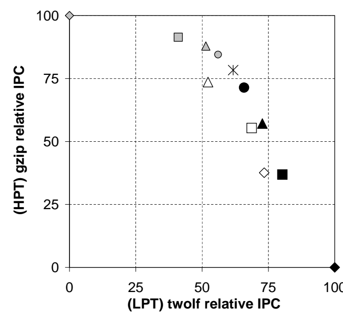
- MB-ILP: this case is the opposite to the previous one. Given that the total throughput comes largely from the LPT that is ILP, when it is denied many resources, its speed degrades fast and total throughput is degraded as well.



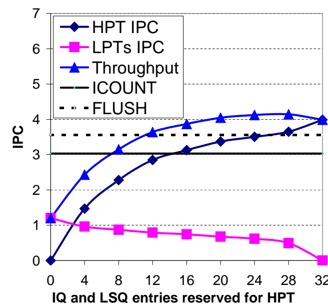
(a) QoS space for ILP-ILP workload



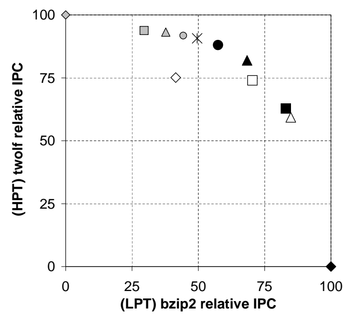
(b) IPC values and overall throughput



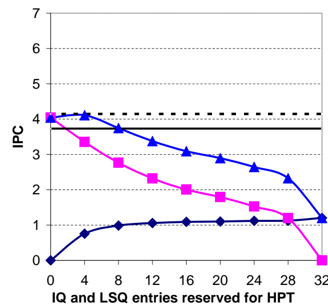
(c) QoS space for ILP-MB workload



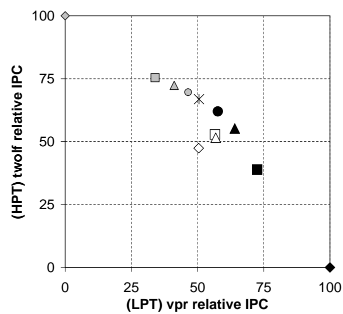
(d) IPC values and overall throughput



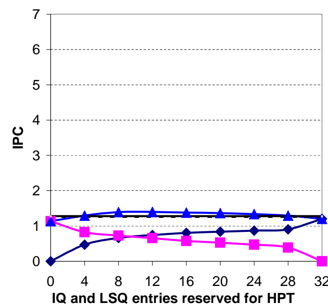
(e) QoS space for MB-ILP workload



(f) IPC values and overall throughput



(g) QoS space for ILP-ILP workload



(h) IPC values and overall throughput

Fig. 4.9: QoS space, IPC values, and overall throughput for all workload types

- MB-MB: throughput shows a flat curve that is about the same as for the *icount* and *flush* fetch policies as was the case for the ILP-ILP workload. Resources taken from one thread can effectively used by the other thread.

We conclude that by controlling resource allocation we can navigate through the QoS space and bias the execution of a workload to a prioritized thread. At the same time, we still reach considerable throughput for the LPT. Hence, our proposal to provide QoS in an SMT by means of resource allocation is a feasible approach.

4.3.3 Conclusions

We have approached the collaboration between OS and SMT as Quality of Service management, where the SMT processor provides 'levers' through which the OS can fine tune the internal operation of the processor in order to meet certain QoS requirements, expressed as points or areas in the QoS space. These levers are provided by making an explicit resource allocation of most critical shared resources in the SMT.

4.4 Chapter summary

In this chapter we have made a study of the benefits of using explicit resource allocation in SMT processors. We have shown, by evaluating two simple mechanisms, that it is possible to increase the throughput of an SMT processor as well as to influence to a great extend the execution of a thread in a workload, so that the OS can reach a large part of the QoS space. These mechanisms are a first step toward showing the benefits that explicit resource allocation have on SMT processors. In next chapters, we explain more sophisticated mechanisms based on the concept of explicit resource allocation.

5

QoS for High-Performance Systems with Explicit Resource Allocation

CHAPTER 5

QOS FOR HIGH-PERFORMANCE SYSTEMS WITH EXPLICIT RESOURCE ALLOCATION

In this chapter we propose a mechanism whose main aim is to improve throughput and fairness of SMT processors. Unlike, our first two methods proposed in Chapter 3, this mechanism is based on the concept of explicit resource allocation.

5.1 Dcra

In current SMTs, resource distribution among threads is either static or fully dynamic. A static resource distribution (used, for example, in the Pentium 4 [47]) 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, the fetch policy 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 [20][43] 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 [63], 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 section, 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 section.

In order to control the amount of resources given to each thread, we use the concept of a *resource allocation policy* introduced in Chapter 4. This direct control allows a better use of resources, reducing 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 that the SMT processor obtains.

In this section, 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) [26][47][50] and also the best dynamic resource-conscious fetch policies like flush++ [13] in both throughput and fairness [46]. 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 it presents a better throughput-fairness balance.

5.1.1 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.

Dcra first dynamically classifies threads based on the execution phase they are in, high ILP or memory-bounded. Next, we determine which resources are being used by each thread in the phase it is in. After that, dcra uses a sharing model to allocate resources to threads based on the classification previously made. Finally, the sharing model also ensures that threads do not exceed their allocated resources.

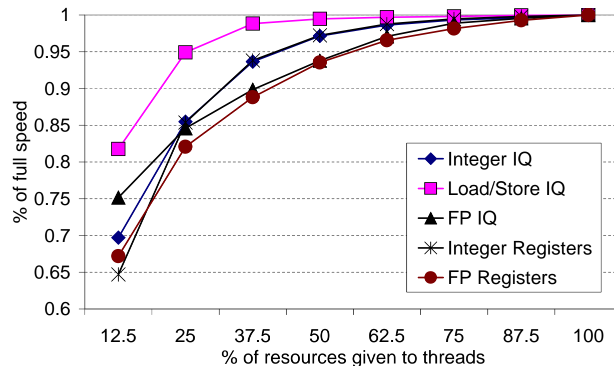


Fig. 5.1: Average IPC of SPEC benchmarks as we vary the amount given to them when the data L1 cache is perfect.

Our model bases on the fact that threads without outstanding L2 misses require fewer resources to exploit ILP than threads with L2 misses. Figure 5.1 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 rename registers, 32-entry issue queues, and the remainder parameters of our baseline configuration. 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.

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,

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

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.

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 into 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 ROB entry and, many of them, a physical register. Also, all instructions depending on the missing load hold an 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.

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.

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 [21], 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.

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 (5.1), where R is the total number of entries of that resource and T is the number of running threads.

$$E = \frac{R}{T} \tag{5.1}$$

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 (5.2), where F is the number of fast threads.

$$E_{slow} = \frac{R}{T}(1 + C * F) \tag{5.2}$$

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

Table 5.1: 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. C=1/T.

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

resource allocation for each type of resource and we 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:

$$E_{slow} = \frac{R}{F_A + S_A}(1 + C * F_A) \tag{5.3}$$

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.

Example: assume a shared resource with 32 available entries in a processor that runs 4 threads and a sharing factor $C = 1/T$. Table 5.1 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.

Implementation of the allocation policy

Figure 5.2 shows the processor modifications required for a possible implementation of our dynamic allocation policy. The modifications focus on two main points:

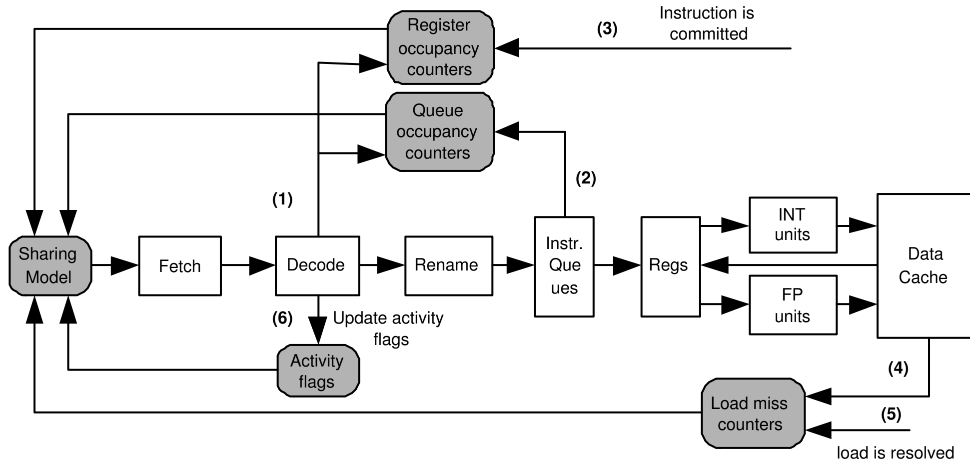


Fig. 5.2: Possible implementation of our dynamic allocation policy

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 5.2). 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 until

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

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 issue queues and one for the physical registers.

5.1.2 Methodology

In Table 5.2 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.

As in previous experiments, programs are divided into two groups based on their cache behavior (see Table 2.2).

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 [27, 31, 65] 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

Table 5.2: Baseline configuration

Processor Configuration	
Pipeline depth	12 stages
Fetch/Issue/Commit Width	8
Issue Queue Entries	80 int, 80 fp, 80 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	352 integer, 352 fp
(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 5.3: Workload classification based on cache behavior of threads.

Number threads	Type	Workload group 1	Workload group 2
2	ILP	gzip, bzip2	wupwise, gcc
	MIX	gzip, twolf	wupwise, twolf
	MEM	mcf, twolf	art, vpr
3	ILP	gcc, eon, gap	gcc, apsi, gzip
	MIX	twolf, eon, vortex	lucas, gap, apsi
	MEM	mcf, twolf, vpr	swim, twolf, equake
4	ILP	gzip, bzip2, eon, gcc	mesa, gzip, fma3d, bzip2
	MIX	gzip, twolf, bzip2, mcf	mcf, mesa, lucas, gzip
	MEM	mcf, twolf, vpr, parser	art, twolf, equake, mcf

(a) First two workload groups.

Number threads	Type	Workload group 3	Workload group 4
2	ILP	fma3d, mesa	apsi, gcc
	MIX	lucas, crafty	equake, bzip2
	MEM	art, twolf	swim, mcf
3	ILP	crafty, perl, wupwise	mesa, vortex, fma3d
	MIX	equake, perl, gcc	mcf, apsi, fma3d
	MEM	art, twolf, lucas	equake, vpr, swim
4	ILP	crafty, fma3d, apsi, vortex	apsi, gap, wupwise, perl
	MIX	art, gap, twolf, crafty	swim, fma3d, vpr, bzip2
	MEM	equake, parser, mcf, lucas	art, mcf, vpr, swim

(b) Last two workload groups.

workloads. We have used the workloads shown in Table 5.3. 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 5.3, 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.

5.1.3 Performance evaluation

We compare our dcra policy with some of the best fetch policies currently published: icount [64], stall[63], flush[63], flush++[13], dg[20] and pdg[20]. Our results show that for the setups examined in this section, 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.

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 [50] 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 running threads.

Figure 5.3 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 5.4 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 5.4. However, dcra also is efficient for ILP and MEM workloads because it also classifies threads according to resource usage.

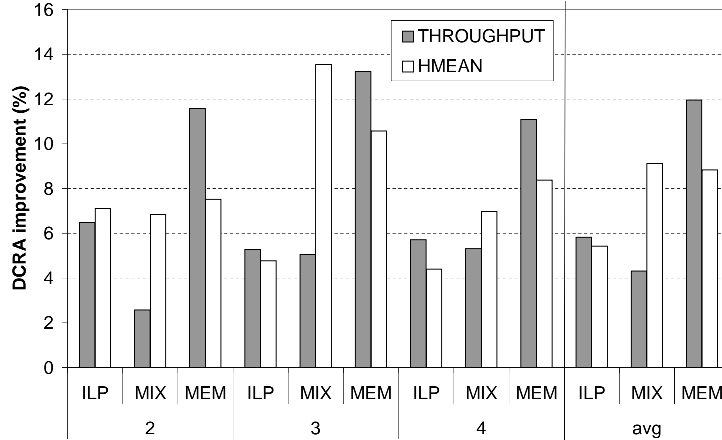


Fig. 5.3: Throughput/Hmean results of dcra compared to static resource allocation

Table 5.4: Distribution of threads in phases for 2-thread workloads

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

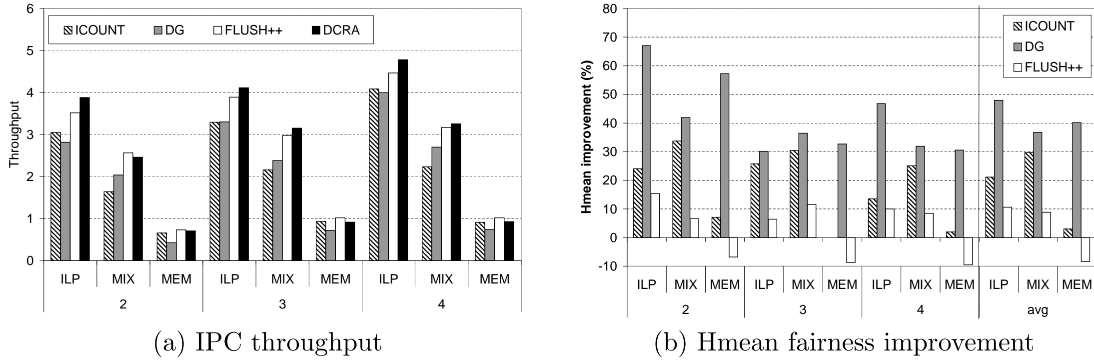
Dcra vs. I-fetch policies

In this subsection, we compare the dcra policy with icount, flush++, and dg.

Figure 5.4(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 5.4(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.



(a) IPC throughput (b) Hmean fairness improvement

Fig. 5.4: Throughput/Hmean improvement of dcra over icount, flush++, and dg.

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 5.1). 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.

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 5.5 shows the average performance improvement of dcra over icount, flush++, dg, and sra, as we change the number of physical registers from 320

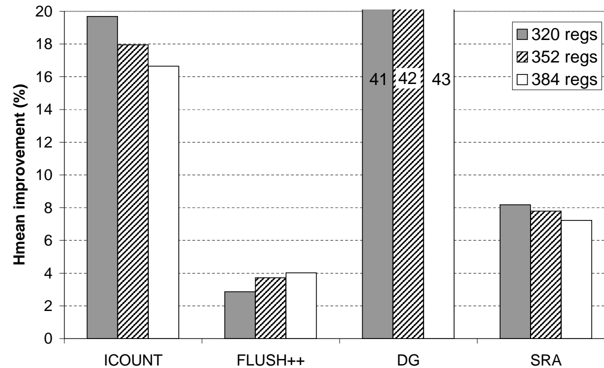


Fig. 5.5: Dcra Hmean improvement over other mechanisms as we change the register pool size.

to 384 entries. For this experiment, we have used 80-entry queues and a memory latency of 300 cycles.

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.

Issue queues: Figure 5.6 shows the average Hmean improvement of dcra over the other design alternatives icount, flush++, dg, and sra, as we increase the number of issue queue entries from 32 to 80 entries. For this experiment, we use 320 physical registers and a memory latency of 200 cycles. We can see that for all policies the trend is the same than for the register file.

Memory latency: As we have seen in this section, 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 5.7 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

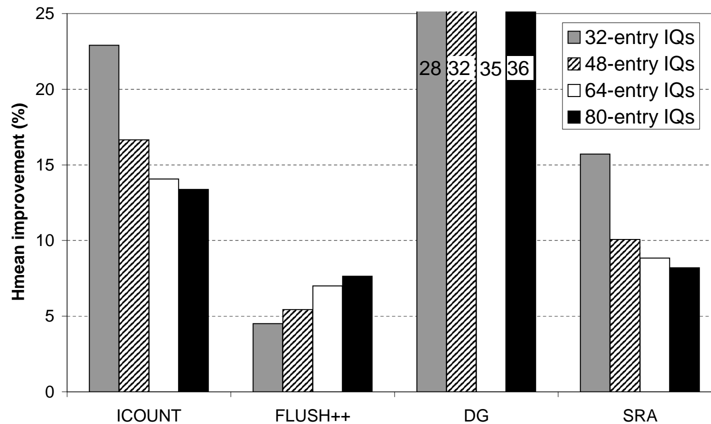


Fig. 5.6: Dcra Hmean improvement over the other mechanisms as we vary the IQs.

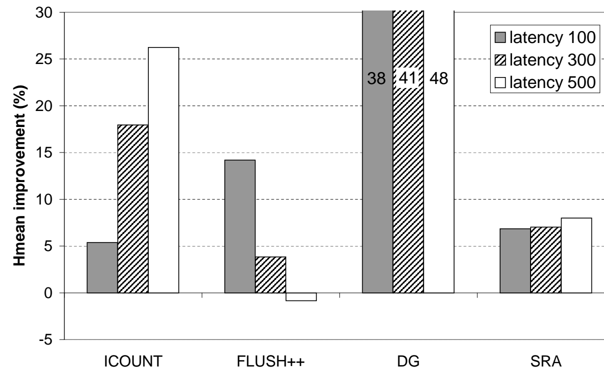


Fig. 5.7: Hmean improvement of dcra over other mechanisms as we change the memory latency.

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.

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.

5.1.4 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. Both situations may degrade the performance of the processor.

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.

5.2 Chapter summary

In this chapter we have proposed a resource allocation policy, dcra, which uses the concept of explicit resource allocation to improve performance. We would like to emphasize that dcra is just an example of resource allocation policy. Depending on the particular hardware resources of the processor under consideration this policy should be changed. Even though, new proposals will be based on the concept of explicit resource allocation.

6

QoS for Real-Time Systems

CHAPTER 6

QOS FOR REAL-TIME SYSTEMS

In this chapter we propose a new mechanism needed in order to enable SMTs to be used in real-time environments. This mechanism involves changes in both the SMT hardware and the real-time OS job-scheduler. We start by analyzing the requirements that a real-time system can request from an SMT processor. After that, we analyze how previous work has dealt with these requirements. Finally, we present two proposals to deal with these requirements.

6.1 Introduction

Real-time systems, especially real-time embedded systems, have specific constraints and characteristics, such as time constraints, low-power requirements, and severe cost limitations, that differentiate them from high-performance systems. Processors for embedded systems typically are simple, with short pipelines and in-order execution. When they are used for real-time applications, they also lack unpredictable components, such as caches and branch predictors. These bare processors provide predictable performance, and hence they can guarantee worst-case execution times of real-time applications. However, embedded systems must host increasingly complex applications and have increasingly higher data throughput rates. To meet these growing demands, future embedded processors will resemble current high-performance processors. For example, the new Philips TriMedia already has a deep pipeline, L1 and L2 caches, and branch predictors [28]. But because of their unpredictable components, such processors have unpredictable execution times, so they are difficult to use in real-time applications. Usually embedded processors are required to be low in cost, hence obtaining as much performance as possible from each resource is desirable. A viable option is an SMT processor, which shares many resources between several threads for a good cost-performance tradeoff [41].

Resource sharing approaches like SMTs allow threads to share the cache hierarchy, which reduces their energy requirements for executing applications: the fraction of the area devoted to caches grows in every new processor generation, mainly due to the L2 cache. Thus, even though low leakage transistors are used for caches, their large area makes them to be the one of the leakiest structures in the chip. Some authors indicate that this leakage will represent more than 50% of all the leakage of the processor [42]. Other authors show that static consumption grows faster than the dynamic comsumption, achieving up to 40-50% of the processor

consumption [24][30][69]. Hence, the L2 cache could consume 25% of the total processor consumption. Thus, sharing the L2 cache among applications, as SMTs do, alleviates this overhead, reducing the power required to execute applications.

Other resource sharing approaches include multiprocessors, which share only the higher levels of the memory hierarchy. The main point in multiprocessors is that different threads are assigned to separate processors, hence their performance is more predictable. However, multiprocessors share few resources between threads, and hence their cost-performance ratio is worse than that of SMTs.

Also, in a limited form of SMT, different threads share only a few resources, typically the instruction and data caches and the functional units, but each thread has its own instruction pipeline. An example is the Meta processor [41]. This solution has better performance predictability for the individual threads but may underuse many resources, thus reducing total throughput. Hence, the cost-performance tradeoff is worse than that of a full-fledged SMT.

Despite these advantages of SMT processors there are two main reasons that make the use of SMTs in real-time systems difficult.

- First, in the traditional collaboration between the OS and the SMT, the OS only assembles the workload, whereas the processor decides how to execute this workload. Hence, part of the OS's traditional responsibility has "disappeared" into the processor. Consequently, the OS cannot guarantee time constraints on the execution of a thread if that thread must run concurrently with other threads, even though the processor has sufficient resources to run threads concurrently. To handle this situation, the SMT processor should be able to guarantee specific requirements set by the OS. This implies a tight interaction between the OS and the processor, so that the OS can exercise more control over how threads execute and how they share the processor's internal resources.

- Second, current SMTs lack flexibility providing quality of service since they are designed with the main objective of increasing throughput: the fetch policy of the SMT decides how instructions are fetched from the threads, thereby implicitly determining the way internal processor shared resources are allocated to the threads.

QoS requirements such as throughput or fairness might be acceptable in many systems. However, one can easily imagine situations where the OS may need to impose additional requirements, i.e., the OS may require that a designated thread runs at a minimum performance in order to meet some real-time constraints. In an uniprocessor system, such a requirement could be achieved guaranteeing that over a period of time, the designated thread would be allocated an amount of CPU time, which proportional to the performance requested. However, this approach would not be useful in an SMT. When the designated thread is running, it will typically be sharing processor resources with other threads (if it was running on its own, it

would clearly underutilize the SMT and defy the whole point about having SMTs). As a result, the designated thread's performance requirement cannot be guaranteed because the fetch policy used for resource sharing would assign resources in order to meet its own overall objectives rather than those of the OS. If it would serve its purpose to allocate fewer resources to the designated thread, it would do so.

To deal with this situation, the OS should be able to exercise more control over how threads are executed and how they share the processor's internal resources. In other words, the hardware should provide the OS with some kind of QoS that can be used by the OS to better schedule jobs. Thus, if we want to be able to control the speed of a particular thread on an SMT, both the traditional OS-level job scheduler and current SMT approaches to resource sharing by means of Ifetch policies are no longer adequate and a closer interaction is required.

In this chapter we present two different approaches that use the idea of explicit resource allocation to solve this problem: Predictable Performance or PP (Section 6.3) and Low-Variability Performance or LVP (Section 6.4). The main characteristics of both approaches are the following. PP achieves better results that LVP but it is more complex to implement. In addition PP can be applied only if applications under consideration have a number of characteristics that we discuss in more detail in the following sections.

6.2 Related work

In [64] it is observed that the total throughput of an SMT processor is highly sensitive to the instruction fetch policy. Other researchers have suggested policies to improve the usage of SMT resources in cases where a thread is stalled as a result of a cache miss or a mispredicted branch [13][20][43][57][63]. All these fetch policies try to optimize total IPC and/or reduce energy consumption. They do not allow control of how threads are executed to meet particular requirements, in contrast to the approach adopted in this section.

To the best of our knowledge, there does not exist much work on real-time constraints for SMT architectures.

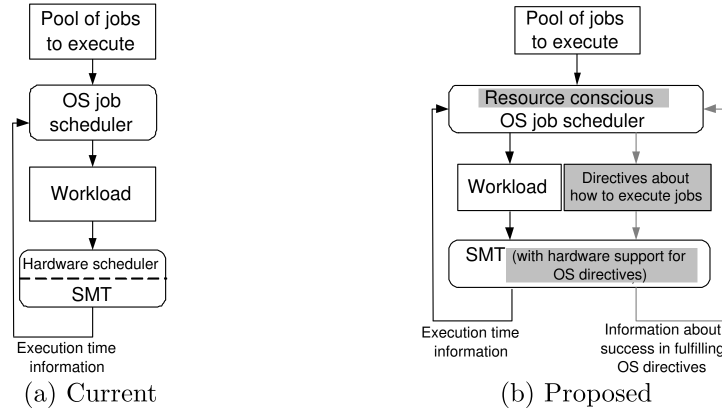
Jain *et al.* [34] study soft real-time scheduling for SMT: they mainly focus on how specific workloads can be assembled from a pool of tasks that is larger than the number of available contexts. Therefore, they address the so-called *co-scheduling* problem for SMT processors. This chapter concentrates on the resource *sharing problem*. This problem consists in determining how internal resources of the processor should be allocated to a given workload in order to guarantee a certain quality of service requirement. In [34], authors briefly discuss the resource sharing problem as well. The authors propose a method to solve the problem of low predictability of SMTs that profiles all possible combinations of tasks. By comparing the IPC of a thread when it is executed in a given workload, IPC_{SMT}, with the IPC that the thread achieves when it is run in isolation, IPC_{alone} (full speed), the slowdown that the thread suffers from being executed in a context is determined. This information

is given as additional input to the scheduler so that the lack of predictability is erased as it knows the IPC of each thread under all possible scenarios or workloads. In addition, this information is used to increase performance since the scheduler selects those scenarios that lead to the highest *symbiosis* among threads and thus the highest performance. The main drawbacks of this solution are two. First, is the prohibitively large number of profiles required. For a task set of K tasks and a target processor of N contexts, we have to profile all $\frac{N!}{K!(N-K)!}$ possible combinations. And second, we may not know beforehand which threads will execute.

A similar solution is proposed in [59]. Authors propose several OS level job schedulers to enforce priorities. Mostly, these schedulers find co-schedules from a pool of runnable jobs that is larger than the number of hardware contexts. Their *SOS* policy runs jobs alone on the machine to determine their full speed, runs several job mixes in order to determine the best mix that exhibits symbiosis, and finally runs jobs alone in order to meet priorities. This approach may under-utilize the machine resources since in many time frames jobs are running alone, in contrast to our approach in which the mix almost always runs together. Next, they propose an extension to the *icount* fetch policy by including handicap numbers that reflect the priorities of the jobs. This approach suffers from the same shortcomings as the standard *icount* policy, namely, that resource management is implicitly done by the fetch policy. Therefore, running times of jobs are still hard to predict, rendering this approach unsuited for real-time constraints. For example, high priority memory bounded threads will clog the pipeline after L2 misses, even more than is the case in standard *icount*. We will show a real example of this situation in the next section.

Dorai and Yeung [18][19] propose transparent threads, which is a mechanism that allows background threads to use resources that a foreground thread does not require for running at almost full speed. Their proposal does not allow the foreground thread to run at a given percentage of its full speed as is the case in our proposal. In a certain sense, this work addresses the problem of job prioritization from the opposite side as we do: whereas we propose mechanisms to assign resources to the High Priority Thread in order to meet constraints, they propose mechanisms to utilize resources by background threads that are left over by the foreground thread. Since they only solve the problem of running a foreground thread at its full speed, their approach is much less flexible than ours.

In [9], the authors propose an approach where the WCET is specified assuming a virtual simple architecture (VISA). At execution time, a task is executed on the actual processor. Intermediate virtual deadlines are established based on the VISA. If, during execution, a task fails to meet its intermediate deadlines, the processor is reconfigured to implement the VISA, bounding the execution time of the task. If the actual processor is an SMT and a task fails to meet its intermediate deadlines, the SMT is switched to single-threaded mode, to ensure that tasks can meet their deadlines. The authors conclude that fetch policies that attempt to maximize throughput, like *icount*, should be “balanced” for minimum forward progress of real-time tasks. This is precisely the target of our mechanisms: we ensure a minimum



(a) Current (b) Proposed

Fig. 6.1: OS/SMT collaboration

amount of resources for a given time-critical thread so that it meets its deadline regardless of the other threads executed in its workload. Our approaches are orthogonal to the VISA framework: a time-critical thread is executed on the actual SMT processor that provides the thread with a given percentage of resources. In the event that the task does not meet its intermediate deadlines, instead of switching the processor to single-threaded mode, we can increase the amount of resources given to it, so that the task meets its deadline and its overall performance does not drop drastically.

6.3 Predictable Performance

In this section, we present novel architectural support for OSs on SMT processors that enables a close interaction between both. This is used to guarantee that jobs in a workload can achieve a certain performance requirement. As opposed to traditional monolithic approaches where the OS has no control over the resource sharing of an SMT architecture (see Figure 6.1(a)), our approach is based on a resource sharing policy that is continuously adapted as it needs to take into account OS requirements (see Figure 6.1(b)). The advantages of the proposed approach are two-fold.

First, the OS can control the execution of jobs at any given time. The OS selects a workload consisting of several threads and indicates to the processor that certain threads should be considered as *Predictable Performance Threads* (PPTs) and must execute at a certain percentage of their full speed, that is, the speed that a thread can achieve when it is run alone on the processor. We have experimented with up to 2 PPTs using our mechanism. When 1 thread is required a given IPC, we obtain successful results, with an error lower than 2%, for target percentages ranging from 10% to 80%. When there are 2 PPTs, we run both jobs exactly at the required percentage for a broad range of percentages. For another broad range of required percentages, our mechanism can realize the target for the thread with the highest priority and give a close approximation of the required speed for the thread with

the second highest priority. This degradation only arises when there are insufficient number of resources inside the processor to realize both requirements together.

Second, we show that the remaining *Unpredictable Performance Threads* (UPTs) can make good use of the resources that are not needed by the PPTs. As a result, our prioritization mechanism not only maintains total throughput, but is in fact capable of outperforming traditional fetch mechanisms in this respect. Thus, a resource conscious scheduler, by using our resource allocation mechanism, can perform better than a traditional OS job scheduler using a fetch policy as a single solution for all cases. In fact, our mechanism achieves at least 90% of the performance of one of the best currently known fetch policies for SMTs, like flush++ [13].

6.3.1 Problem statement

As illustrated in the introduction, the approaches currently employed for resource sharing in SMTs, I-fetch policies, may disregard any target set by the OS. This is a consequence of the additional level of scheduling decisions, introduced by resource sharing inside the SMT, which has been largely examined independently of the more traditional OS level *co-scheduling* phase.

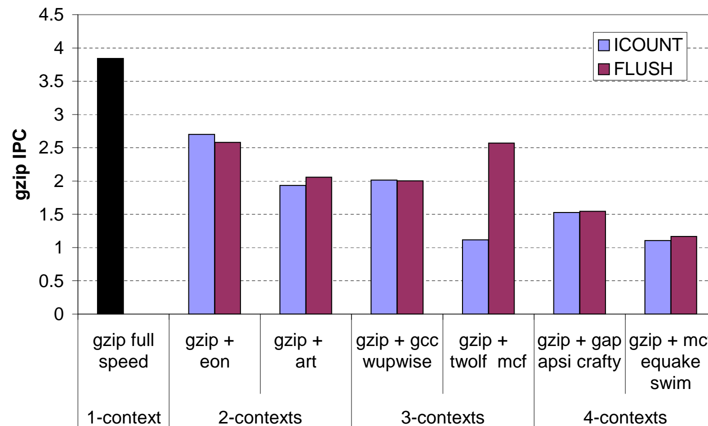


Fig. 6.2: IPC of `gzip` for different contexts and different fetch policies

In [9], the authors conclude that current fetch policies that attempt to maximize throughput, like *icount*, should be “balanced” for minimum forward progress of real-time tasks otherwise real-time tasks may miss their deadline. In Figure 6.2 we show an example of how the performance of an application varies depending on the workload it is executed in when it is run in an SMT. Figure 6.2 shows the IPC of the `gzip` benchmark when it is run alone (full speed) and when it is run with other threads using two different fetch policies, *icount* [64] and *flush* [63]. As we can see, its IPC varies a lot, depending on the fetch policy as well as the nature of the other threads running in the context. This is caused by the fact that management

of resources (IQ entries, registers, FUs, etc.) is not explicit. This shows that the current collaboration between the OS and the SMT hardware, in which the OS has no control over the resource sharing of an SMT architecture, does not provide control over the execution of applications.

To the best of our knowledge, only a few papers have identified the need for a closer interaction between SMT co-scheduling and resource sharing algorithms in order to force priorities.

In [59], an extension to the *icount* fetch policy is proposed by including handicap numbers that reflect the priorities of jobs. This approach suffers from the same shortcomings as the standard *icount* policy, namely, that resource management is implicitly done by the fetch policy. Therefore, although this mechanism is able to prioritize threads to some extend, running times of jobs are still hard to predict, rendering this approach unsuited for real-time constraints. For example, in Figure 6.3, we show the IPC of the `gzip` benchmark for different handicap numbers and workloads. We observe that running `gzip` with the same handicap leads to different IPC values, with a variation of up to 60% depending on the workload. Hence, even when the *icount* handicap prioritization mechanism achieves some prioritization of threads, it is still far from the objective of this section, namely, ensuring that several threads in a workload run at a given target IPC.

Finally, the Power5 [36] processor uses a mechanism to control the decode bandwidth. In our architecture this control provides the same results as controlling the fetch bandwidth. However, given that no information on the internal resource allocation of the Power5 has been released, we cannot compare the efficiency of this mechanism and ours.

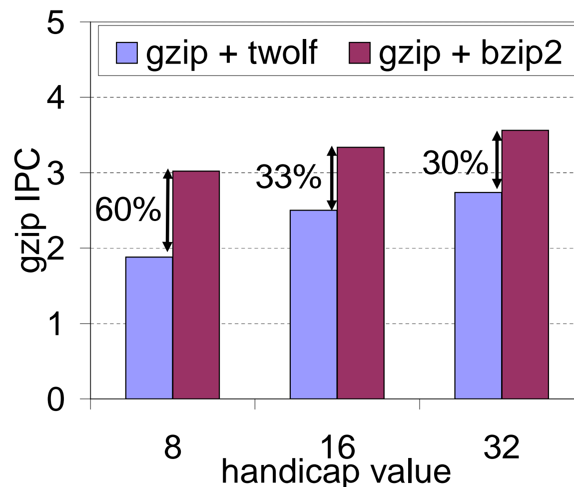


Fig. 6.3: IPC of the *gzip* benchmark for different handicap values

6.3.2 QoS problem definition

In this section, we move a step further than existing work and we propose a novel approach for a *dynamic* interaction between the OS and the processor which allows the former to pass specific requests onto the latter. In particular, we focus on the following challenge: given a workload of N applications[1] and one or two Predictable Performance Threads (PPT0 and PPT1) in this workload, where PPT0 has a higher priority than PPT1, find a resource sharing policy to:

- Ensure that PPT0 runs at (at least) a given target IPC that represents $X\%$ of the IPC it would get if it were to be executed alone on the machine.
- Ensure that PPT1 (if PPT1 is given) runs at a given target IPC that represents $Y\%$ of the IPC it would get if it were to be executed alone on the machine, or as close as possible to this $Y\%$ when there are insufficient resources to realize the targets for both PPT0 and PPT1 at the same time.
- Maximize the throughput for the remaining $N-1$ (or $N-2$ if a PPT1 is given) Unpredictable Performance Threads (UPTs) in the workload.

To tackle a challenge such as the one described above we use a generic approach to resource sharing for SMTs, which addresses such challenges as a QoS problem. As explained in section 4.3, this approach is inspired by QoS in networks in which processes are given guarantees about bandwidth, throughput, or other services. Analogously, in an SMT resources can be reserved for threads in order to guarantee a required performance. Our view is that this can be achieved by having the SMT processor provide 'levers' through which the OS can fine tune the internal operation of the processor as needed. Such levers can include prioritizing instruction fetch for particular threads, reserving parts of the resources like IQ entries, etc.

In order to measure the effectiveness of a solution to a QoS problem, we have used the notion of *QoS space* introduced in section 4.3. On an SMT processor, each thread, when running as part of a workload, reaches a certain percentage of the speed it would achieve when running alone on the machine. Hence, for a given workload consisting of N applications and a given instruction fetch policy, these percentages give rise to a point in an N-dimensional space, called the QoS space [2]. For example, Figure 6.4 shows the QoS space for two threads, `eon` and `twolf`. In this figure, both x- and y-axis span from 0 to 100%. We have used three fetch policies: *icount* [64], *flush* [63], and data gating (*dg*) [20] in our baseline configuration. Theoretically, if a policy leads to the point (x, y), then it is possible to reach any point in the rectangle $(0, 0), (x, y)$ by judiciously inserting empty fetch cycles. Figure 6.4 also shows a more general picture in which the dashed curve indicates points that intuitively could be reached using some fetch policy. Obviously, by assigning all fetch slots and resources

[1] We assume throughout the section that the workload is smaller than or equal to the number of hardware contexts supported by the processor.

[2] The notion of QoS space is applicable to other quantities, not only percentage of IPC.

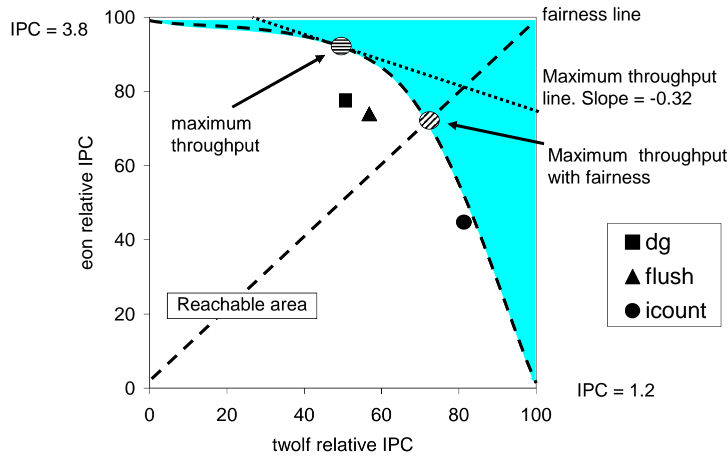


Fig. 6.4: QoS space for three fetch policies and important QoS points and areas

to one thread, we reach 100% of its full speed. Conversely, it is impossible to reach 100% of the speed of each application at the same time since they share resources.

In Figure 6.4 we see that the representation of the QoS space also provides an easy way to visualize other metrics used in the literature. Points of equal throughput lie on a single line whose slope is determined by the ratio of the maximum IPCs of each thread (in this case, $-1.2/3.8 = -0.32$). Such a point with maximum throughput is also indicated in the figure. Finally, points near the bottom-left top-right diagonal indicate fairness, in the sense that each thread achieves the same proportion of its maximum IPC. In either case, maximum values lie on those lines that have a maximum distance from the origin.

6.3.3 Solution of the QoS problem

The notion of QoS implies the existence of hardware mechanisms in the SMT processor that can offer effective control over the execution of threads. It also implies that the OS should have some knowledge of the full speed of jobs in order to be able to give requirements to the hardware, such as, "execute at a certain percentage of the full speed".

Our mechanism requires that all instances of an application have more or less the same IPC. Fortunately, it has been shown [33] that media applications have these properties so that all approaches discussed in this section can be applied to multimedia applications. In the challenge considered in this section, we want to guarantee a certain speed for the PPTs. We obtain this by reserving the proper resources for them, proceeding hierarchically. First, we reserve resources for PPT0 as it has the highest priority. Then we reserve resources for the subsequent lower in priority, PPT1, from those resources not used by PPT0, and so on. Finally, the remaining resources are given to the UPTs.

The QoS mechanism we propose combines three key ideas. First, we monitor

characteristics and execution progress of each thread and give feedback to the OS. Second, we employ resource administration and allocation, guided by the demands of threads and the external objective given by the OS. Third, we shield PPTs against destructive interference of the other threads.

Another key point in our mechanism is that programs experience different phases in their execution in which their IPC varies significantly. Hence, if we want to realize a certain percentage of the full speed of a program, we need to take into account this varying IPC. We illustrate this by an example. Figure 6.5 shows local IPC values for `gap` for a period of 4.5 million cycles in which each value has been determined over an interval of 15,000 cycles. Assume that the OS requires the processor to run this thread at 80% of its full speed. The solid line is the average IPC for this period and the dashed line represents the value to be achieved by the processor. It is easily seen that during some periods it is impossible to achieve this 80% of the global IPC, even if the thread were given all the processor resources. Moreover, if the processor achieves this 80% of the global IPC during the first part of the interval and subsequently gives all resources to this thread to achieve full speed during the second part, then the overall IPC value it would realize would be lower than 80% of the global IPC.

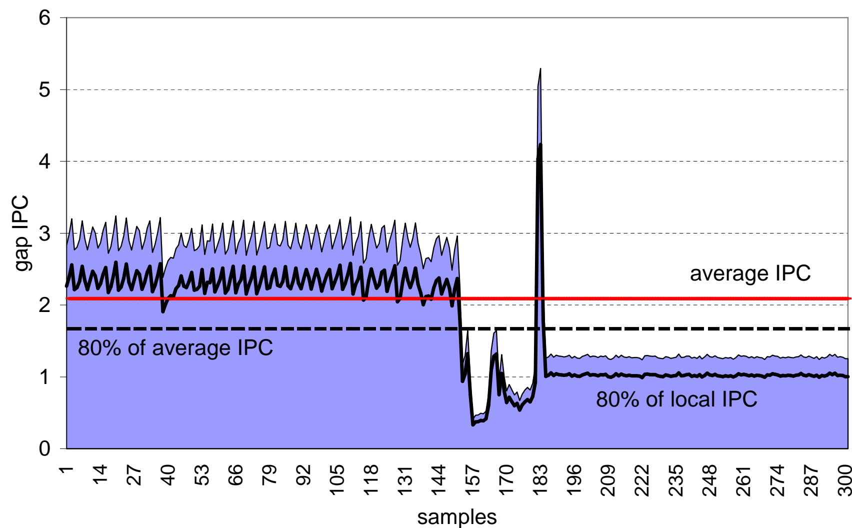


Fig. 6.5: Local IPC of the `gap` benchmark

The basis of our mechanism for dynamic resource allocation rests on the observation that in order to realize $X\%$ of the overall IPC for a given job, it is sufficient to realize $X\%$ of the maximum possible IPC *at every instant* throughout the execution of that job. This is illustrated in Figure 6.5 by the bold faced curve labeled "80% of local IPC". Hence, the mechanism needs to determine the variations in IPC of the PPT. In order to do this, we distinguish between two phases in our proposed mechanism that are executed in alternate fashion, as shown in Figure 6.6.

- During the first phase, the *sample phase* (60,000 cycles), all shared resources are given to a PPT and the other threads are temporarily stopped. As a result, we obtain an estimate of the current full speed of that PPT during this phase which we call the *local sampled IPC*.
- During the second phase, the *tune phase* (1.2M cycles), UPTs are allowed to run. Our mechanism dynamically varies the amount of resources given to both PPTs to achieve the *local target IPC*. It is given by the local sampled IPC computed in the last sample period times the required percentage given by the OS, which we call *target percentage*.

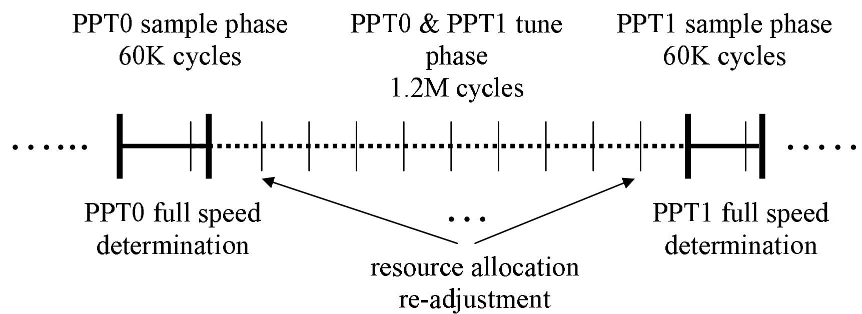


Fig. 6.6: Sample and Tune phases

Clearly, if we are able in the sample phase to measure reasonably accurately the full speed of a given PPT and in the tune phase to realize a percentage $X\%$ of that sampled IPC, then we obtain an overall IPC that is about $X\%$ of the IPC of that PPT would have when executed alone in the processor. In the next two subsections we discuss the sample phase and the tune phase in more detail.

Sample phase: Determining the local IPC of a PPT

During the sample phase, we determine the local IPC of a PPT by giving it all shared resources and hence suspending the others momentarily. Note that the longer the sample phase, the longer the time that the SMT is dedicated to only one thread, reducing its overall performance and starving the Unpredictable Performance Threads. Hence, we have to determine the local IPC of the PPT thread as fast as possible. Our mechanism dedicates only 5% of the total execution time to the sample phase.

In our simulated architecture there are the following shared resources: fetch and issue slots, IQ entries, physical registers, caches, TLBs and the branch predictor. Our mechanism takes into account the following: L2 cache, IQs, physical registers, and the fetch and issue bandwidth. The rest are *freely* shared among all threads. These resources are allocated in the following way:

First, the allocation of the IQs and the physical registers is as follows: depending on the particular needs of the PPTs during their execution, the number of resources

allocated to them is changed. When a shared resource is partitioned, one part is dedicated to PPT0, another part to PPT1, and the remaining part is dedicated to the UPTs.

Second, fetch and issue bandwidth is shared hierarchically. The PPT0 has priority to use it and when PPT0 cannot use the entire bandwidth, it is given to the PPT1. The remaining bandwidth is used by UPTs, breaking ties with *icount*.

Third, unlike previous resources, caches, TLBs and the branch predictor can suffer destructive interference because an entry given to one thread can be evicted by another thread. In order to get more insight into this interference, we show in Figure 6.7 how many inter-thread conflicts the PPT suffers during a 100,000 cycle-long sample phase, averaged over the entire run of a workload consisting of `twolf` as PPT and `mcf`, `equake`, and `swim` as UPTs. We observe that as the sample phase progresses, the number of conflicts goes toward zero for the instruction cache, data cache, TLB, and BTB. From the figure we conclude that after a *warm-up period* of 50,000 cycles most interference in these shared resources is removed. The branch predictor (PHT) takes much longer to clear: we have measured that it takes more than 5,000,000 cycles before inter-thread misses have disappeared. However, we have also measured that this interference is mostly neutral, giving a small loss in the branch predictor hit rate less than 1%. Hence, we ignore the interference in the branch predictor. The interference in the L2 cache is more serious: it extends for about 1.5 million cycles and gives rise to a significant performance degradation (more than 30% for some benchmarks). This high number of cycles shows that we cannot deal with the interference in the L2 by simply extending the warmup phase. We address this problem below.

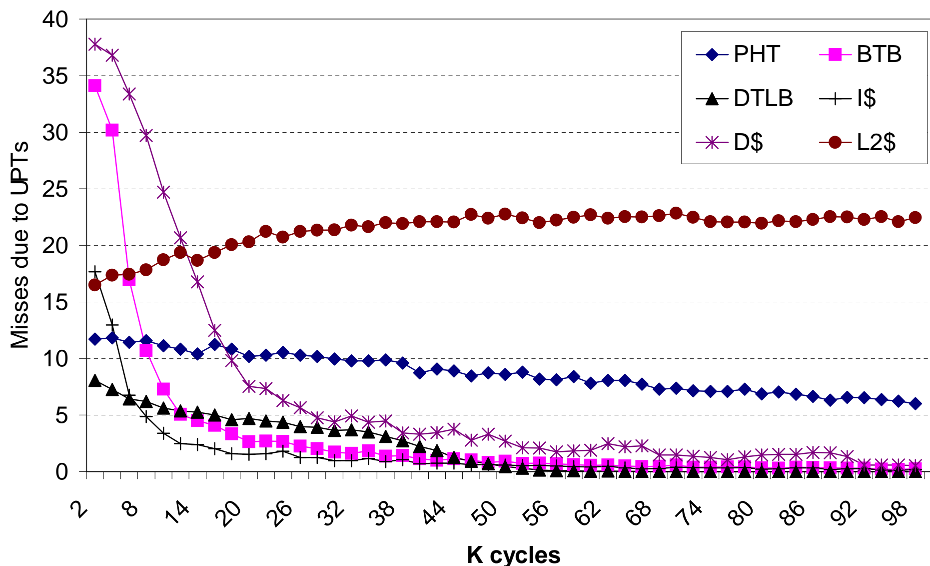


Fig. 6.7: Misses suffered by the PPT due to the UPTs interference

The solution we propose to erase the conflicts in all shared resources, except the L2 cache, consists of splitting each sample period into two sub-phases.

- During the first sub-phase, the *warmup phase*, that consists of 50,000 cycles, the PPT is given all resources but its IPC is not yet determined.
- In the second sub-phase, the *actual-sample phase*, that consists of 10,000 cycles, the PPT keeps all resources and moreover its IPC is determined.

The duration of this phase was determined empirically and is in accordance with the results published in [70]. In this study, the authors present a statistical study showing how to sample the trace of a given thread in order to obtain an accurate measure of its IPC.

Reducing L2 cache misses due to inter-thread interference: The L2 cache is an important source of unpredictability because the effects of threads interferences remain for a long time and because they affect the critical path processor-memory.

Our solution to inter-thread interference in the L2 cache consists of *partitioning* this cache into several parts. One part is dedicated to each PPT and the other part can be used by the entire workload, UPTs and PPTs. In order to meet the demands for varying workloads and program behavior, we employ *dynamic cache partitioning*. We assume that the L2 cache is N-way set associative, where N is not too small: in our simulator, L2 is 8-way set associative. We use a bit vector of N bits that indicates which "ways" or elements in the set are reserved for each PPT. The cache behaves like a "normal" cache, except in the placement and replacement of data. The PPT0 is allowed to use the entire L2, the PPT1 is allowed to use the entire cache except the ways reserved for the PPT0. Finally, the UPTs are restricted to use the remainder subset of all the ways that exist in a set. An extra, most significant, LRU bit is required for each way. This bit is set to one for the reserved ways and to zero for the other ways so that the lines reserved for a PPT always have a higher access count than the lines in the shared part of the cache. The bit of the PPT0 is more significant than the bit of the PPT1.

Let us assume that a load/store instruction misses in the L2 cache and causes a replacement/eviction. If that instruction was issued by an UPT then, only lines belonging to the shared part of the cache are selected for replacement using the LRU algorithm. If it was issued by the PPT1 then, we mask this extra bit and we first select a victim line that belongs to an UPT, if possible. If there does not exist such a line, the LRU line from the lines belonging to the PPT1 and in the shared part is selected as the victim. If that instruction was issued by the PPT0 then, we first select a victim that belong to an UPT. Next, to the PPT1 and finally, if all data in the set belong to the PPT0, the LRU line of the entire set is selected as victim.

Note that this extension allows the cache to be used normally when the SMT does not execute a workload with a designated PPT: the extra bit is always masked. In [17] a different cache partitioning technique called column caching has been proposed. However, this technique addresses a much more general problem of cache

partitioning. Therefore, that technique is too heavy weight to be used for our purposes where the simply mechanism described below suffices.

We propose an iterative method that dynamically varies the number of ways reserved for the PPT. The control is local to the cache and only receives from the processor information about how many, if any, PPTs there are. It also receives information about which phase, sample or tune, these PPTs are currently.

- During each actual sample phase, every time a PPT suffers an intra-thread miss, a per-thread-counter is incremented. We consider that a thread has experienced an intra-thread miss when it is going to use a cache position used by other thread.
- At the end of the sampling period, if the value of the counter is higher than 8 [3], then the number of ways reserved for the PPT is increased by 1.
- If, on contrary, the counter is lower than the threshold, this number is decreased by 1.

In this way, if PPTs experience few L2 misses due to interference, we reduce the number of ways reserved for them. Likewise, if they experience many misses, then we increase the number of reserved ways. The maximum number of ways that each PPT can reserve is an empirically derived value that depends on the configuration and on the relative order of priorities. Of course, the higher the number of ways in the L2 cache, the better this algorithm works. This is not a limiting factor, as current trends in computer architecture show that the number of ways in the outer cache level of processors is increasing. For example, the Sparc64 VI [38] has a 12-way L2 cache. The L2 of the IBM Power5 [36] is 10-way. The AMD K8 has a 16-way L2 cache [2]. In our configuration, we can reserve up to 4 ways for the PPT0 and up to 2 for the PPT1.

Tune phase: Realizing the target IPC

After each sample phase of 60,000 cycles, there is a tune phase where we try to achieve the target percentage of the local IPC measured in the previous sample period, that is, the *local target IPC*. As stated before we want the sample phase to be at most 5% of the total execution time. For this reason each tune phase takes 1.2 million cycles.

We proceed as follows. First, we adjust the partitioning of the L2 cache that remains the same for the entire tune phase. This adjustment has been described in the previous section. Second, the amount of the other shared resources (IQ entries and the physical registers) dedicated to each PPT is dynamically varied as follows.

[3] The value of this threshold has been determined empirically based on the memory latency and the duration of the sample phase. A change of any of these parameters requires an adjustment of this value.

- Each tune phase is split into 80 sub-phases of 15,000 cycles.
- At the end of every sub-phase, the average IPC of the PPTs in this sub-phase is computed.
- If the IPC of the PPT0 is lower than its local target IPC, then the amount of resources given to it is increased. We proceed similarly with the PPT1.
- Otherwise, if this IPC is higher than the local target IPC, then the amount of resources given to this PPT is decreased.

We vary the number of instances of each resource dedicated to a PPT by a fixed amount that equals the total number of instances of that resource divided by a *granularity factor*, which has been set to 16. For example, for the 32-entry issue queues this amount is $32/16 = 2$.

Finally, we have observed that the local sampled IPC values are sometimes lower than they should be due to interference from LPTs, mainly in the L2 cache. This results in local target IPCs that are lower than they should be, and thus, the final IPC obtained for the PPTs is also lower than the target IPC given by the OS. In order to counteract this effect, we also take into account the *global IPC* of the PPT: at the end of each sub-phase we check whether the total IPC of the PPT under consideration up to this cycle is lower than the target IPC given by the OS. We introduce a *compensation term C* for this effect, as shown in formula (6.1). This term artificially increases the local target IPC so that the final IPC of that PPT converges to the target IPC given by the OS. In this formula, X is the target percentage.

$$local\ target\ IPC = \frac{(X + C)}{100} \times local\ IPC \qquad (6.1)$$

Initially C is zero. If the total IPC is smaller than the target IPC, we increase C by 5. On the other hand, if the global IPC is larger than the target IPC, we decrease C by 5. However, we stipulate that C does not become smaller than zero or greater than $100 - X$.

Hardware Implementation

The implementation of our mechanism involves three main parts: establishing the maximum amount of resources that threads can use, tracking the resources used by each thread, and tracking the phase and sub-phase in which the mechanism is in. The implementation we propose is valid for any number of PPT. For simplicity, we explain the case where there are 2 PPTs.

Resource limits: In our architecture, the structures under control are the physical registers and the IQs[4]. We employ two 3-entry tables, one for the IQs and one for the registers, that we call IQlimit and REGlimit. Each entry holds the maximum number of entries that a thread can use of this type of resource. These tables are read every cycle by the threads fetching instructions that cycle. If a thread has

[4] The L2 cache also need to be changed. These changes has been already explained.

allocated more entries than assigned to it for any resource, it is not allowed to fetch more instructions. These tables are written at the end of the sample phase and the tune sub-phases.

Tracking resources: In order to track the number of IQ entries and physical registers used by each PPT and the UPTs, we use 5 counters for each: 3 of the counters track the number of entries used in each IQ, the last two the number of integer and fp registers. In the decode stage, we can determine the IQ type and the physical register, if any, that an instruction is going to use. Hence, all registers are updated in this stage. The IQ counters are decremented when instructions are dispatched. Register counters are decremented when instruction commits.

Tracking phases: at any given time, we have to track the phase/sub-phase our mechanism is in. We use a Finite State Machine (FSM) to do this, as shown in Table 6.1. This FSM is quite simple and can be implemented with four counters and simple control logic.

The FSM starts by establishing the PPT whose full speed we are going to sample in this sample phase, $PPTcurrent$ (S0). All resources are given to that PPT. This is done by simply setting its entries in the IQlimit and REGlimit tables to the maximum number of resources, and resetting the other entries. Next, at the end of the warm-up phase (S2), we begin to compute the IPC of the PPTcurrent. At the end of the actual-sample phase (S4), we compute the local target IPC and set the resource allocation to converge to the local target IPC. We also vary the number of way reserved for the PPTs. At the end of each tune sub-phase, (S6) we vary the resource allocation again. After 80 tune sub-phases we re-start the process by changing the PPT.

Table 6.1: FSM to track phases

State	Description	Next State 1	Next State 2	Actions
S0	Select PPT	S1	-	(1)PPTcurrent= 1-PPTcurrent (2) Init all counters (3)Give all resources to PPTcurrent
S1	Warm-up phase	$counter1 < 50k \rightarrow S1$	$counter1 = 50k \rightarrow S2$	counter1++
S2	Actual Sample Start	S3	-	Start local IPC sampling
S3	Actual Sample phase	$counter2 < 10k \rightarrow S3$	$counter2 = 10k \rightarrow S4$	counter2++
S4	Tune sub-phase Start	S5	-	(1)Compute local IPC and local target IPC (2)Compute L2 reserved ways for PPTs
S5	Tune sub-phase	$counter3 < 15k \rightarrow S5$	$counter3 = 15k \rightarrow S6$	counter3++
S6	Adjust resources	$counter4 < 80 \rightarrow S5$	$counter4 = 80 \rightarrow S0$	(1)counter4++.(2)counter3=0. (3)Adjust resource allocation to achieve local target IPC

6.3.4 Experimental setup

To evaluate the performance of our mechanism, we use a trace driven smtsim simulator derived from smtsim [65] as shown in Chapter 2. Table 6.2 shows the main parameters of the simulated processor, which has a 12-stage pipeline.

The working set of MediaBench benchmarks is less than 10K [25], while the working set of SPEC CPU benchmarks is much bigger 97 Mbytes on average [52]. This increases the number of L2 cache misses and hence the demand for resources of SPEC benchmarks. The present technique is a "proof of concept" of how controlling SMT internal resource can enable SMTs to execute real-time applications. Our purpose is to show the robustness of our method. For this reason, we have used SPEC benchmarks that have high resources demands, which makes it difficult to ensure a minimum IPC for a given PPT. It is clear that if our mechanism works with applications with high resource requirements, like SPEC benchmarks, it will work with applications with less resource requirements, like MediaBench benchmarks.

We do not include workloads with more than 4 threads because several studies [27][31][65] show that for workloads with more than 4 contexts, performance saturates or even degrades because cache and branch predictor conflicts counteract the additional ILP provided by the additional threads.

Table 6.2: Baseline configuration

Processor Configuration	
Fetch/Issue/Commit Width	8
Baseline fetch policy	icount
Baseline fetch architecture	2.8
Issue Queue Entries	32 int, 32 fp, 32 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	320 int, 320 fp
ROB size	256 entries per thread
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, 2-way, 8-bank, 64-byte lines, 10 cycle access
Main memory latency	100 cycles
TLB miss penalty	160 cycles

6.3.5 Results when there is one PPT

In this section we show the results obtained with our strategy. We analyze results when there is only 1 PPT. After that, in the next section we discuss results when there are 2 PPTs at the same time. The results focus on three main points. First, we

show the average performance obtained for the PPT(s) for each workload type. Second, we show the throughput of the UPTs. And third, we show the total throughput obtained.

In this experiment, we consider all combinations in which the PPT is ILP or MEM, and the UPTs are ILP or MEM also. A workload is identified by two parameters: the type of the PPT and the type of the UPTs. For example, a workload of type IM means that the PPT is ILP and the UPTs are MEM . For each of the four workload types, we create four different sets of threads, to avoid that our results are biased toward a specific set of threads, by taking all possible combinations from Table 6.3. That is, we vary the PPT but keep the UPTs fixed in order not to create a prohibitively large number of workloads to examine. We selected as MEM benchmarks those with the highest L2 miss rate (`twolf` and `mcf` for integer benchmarks and `art`, `swim`, `lucas` and `equake` for FP). ILP benchmarks have been selected randomly. In this section, we present average results for each group in each workload. For example, the II set with 3 threads (II3) is the average results of workloads `gzip+(gcc+wupwise)`, `bzip2+(gcc+wupwise)`, `mesa+(gcc+wupwise)`, and `fma3d+(gcc+wupwise)`. For all workloads, the simulation ends when the PPT finishes. Any UPT in the workload that finishes earlier is re-executed.

Table 6.3: Workloads for 1-thread prioritization

	PPT	UPTs
ILP	gzip	eon
	bzip2	gcc,wupwise
	mesa	gap,apsi,crafty
	fma3d	
MEM	twolf	art
	mcf	twolf,mcf
	art	mcf, equake, swim
	lucas	

PPT performance

In Figure 6.3.5 we show, for the different workloads and different target percentages, the overall percentage of full program speed that we have obtained using our mechanism. On the x-axis, the target percentage of the full speed of the PPT is given, ranging from 10% to 90%. For each size of the workload (2, 3, or 4 threads) the achieved IPC for the PPT as a percentage of its full IPC is given. We see that over the entire range of different workloads and target percentages, we achieve this target or a little bit more (approximately 3%). Only on the two extreme ends of the range of targets, we are somewhat off target. We discuss the discrepancies for the 10 and 90% cases, since they give most insight in how our mechanism works.

- 10% case: if the target IPC should be 10% of the full speed, we achieve percentages between 13 and 21. To explain this, first consider the II2 workload

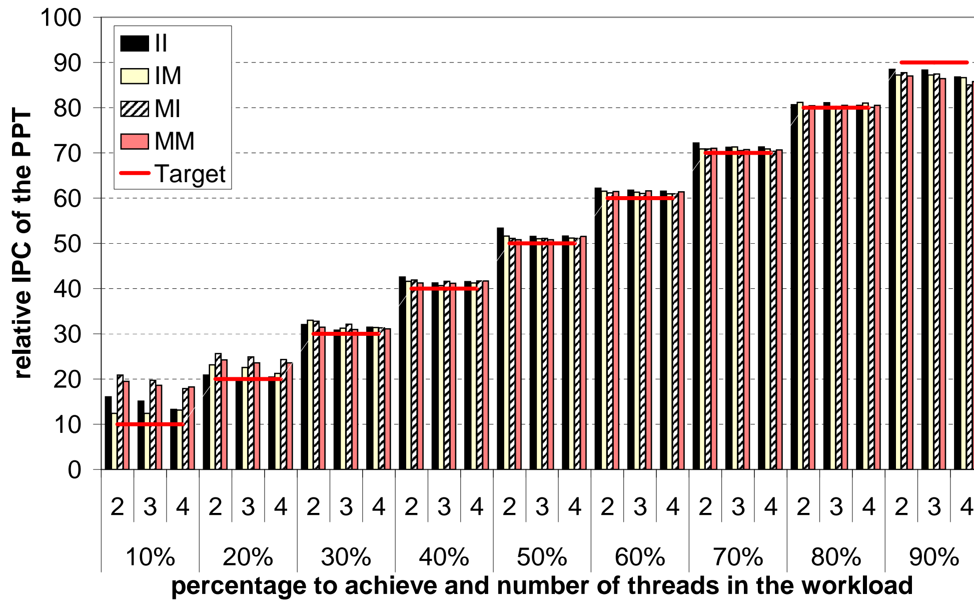


Fig. 6.8: Realized IPC values for the PPT. The x-axis shows the target percentage of full speed of the PPT and size of the workloads. The four different bars represent the four different types of workload discussed in Section 6.3.4

in which two ILP threads are running. Suppose both threads have a full speed of 4 IPC. Then, in 5% of the time during the sample phase, the PPT reaches this full speed. During the remaining 95% of the time, it reaches 0.4 IPC. Hence, in total it reaches $0.05 \times 4 + 0.95 \times 0.4 = 0.58$ IPC which is 15% of its full speed of 4 IPC. Hence, the sample phase that takes 5% of the total running time of the program causes the resulting total throughput of the PPT to be larger than it should be. From Figure 6.3.5 it is also clear that for the workloads II3 and II4 the achieved percentage is closer to 10. This can be explained because, as the number of threads increases, the IPC value of the PPT in the sampling period is lower due to more interference of the other threads. As a result, the overall IPC of the PPT drops a little.

Next, consider the IM workload. In this case, the memory bounded UPTs causes L2 cache pollution, more than is the case for the II workloads. Hence, the measured IPC of the PPT during the sample phase is lower than it should be and during the tune phases the PPT also suffers from interference. Therefore, the effects described for the II case above do not show up as profoundly in the MI case and the overall throughput is closer to 10% as it should be.

For the MI workload, the `mcf` benchmark has a full speed of 0.15 IPC. Hence, 10% of this full speed is only 0.015 IPC. Due to the duration of the sample phase, we reach a slightly higher overall IPC than this. However, the absolute numbers are so small that such a minimal deviation causes a high relative error:

we measured a 30% deviation. Hence, the error in the IPC of `mcf` dominates the average results shown in the figure and therefore the large difference is due to this benchmark. Moreover, in general MEM benchmarks have low IPC values and when they are used as PPT, small differences in their IPCs again cause large relative errors. For the MM workload, the same explanation holds as for the MI workload.

- 90% case: on the other end of the spectrum, when the required percentage is 90, the realized percentage is 2 to 5 percent lower than is should be. To explain these differences, if the UPTs are memory bounded, then they cause much pollution in the L2 cache. Hence, the IPC of the PPT we measure during the sample phases is lower than it should be. Moreover, during the tune phase, memory bounded UPTs cause much interference in the L2 cache also. Therefore, the relative IPC of the IM workloads is lower than the IPC of the II workloads and during the tune phase we achieve an IPC value that is too low also. However, in case the UPTs are ILP, this pollution is much less and therefore, achieved IPC values are higher than for the previous case. We can conclude that when the required percentage is 90 it can be more preferable to run the PPT in isolation and reach 100% of its full speed.

- 30% - 80% case: in more common situations where target percentages range from 30 to 80, we already achieve these percentages almost exactly, being less than 1% over target on average.

Total performance

In Figure 6.9, we show a small variation of the QoS space presented in Section 6.3.2. In this case the y-axis indicates the relative IPC of the unique PPT, and the x-axis the throughput obtained by our mechanism with respect to *flush++* [13]. This fetch policy is an improved version of the *flush* policy proposed in [63]. Results are averaged for workloads with 2, 3, and 4 threads. The legend shows different workload types as well as the relative IPC required for the PPT. For example, the point II20 is the average result when the PPT is ILP and the UPTs are ILP (workload sets II2, II3 and II4), for which the PPT is run at 20% of its full speed. For clarity, we only show those percentages that are a multiple of 20. However, we also show the case of 90% because this high percentage poses particular problems.

- In the II workloads (the triangles in Figure 6.9), all threads are ILP whereby have a high throughput and do not occupy hardware resources for a long time. That means that both, the PPT and the UPTs highly profit resources. As result, for intermediate target percentages, overall performance does not drop much compared to *flush++*. Only for the extreme target percentages this performance is slightly less. On average, our mechanism achieves 90% of the throughput obtained with *flush++* for the II workloads. In addition, we should

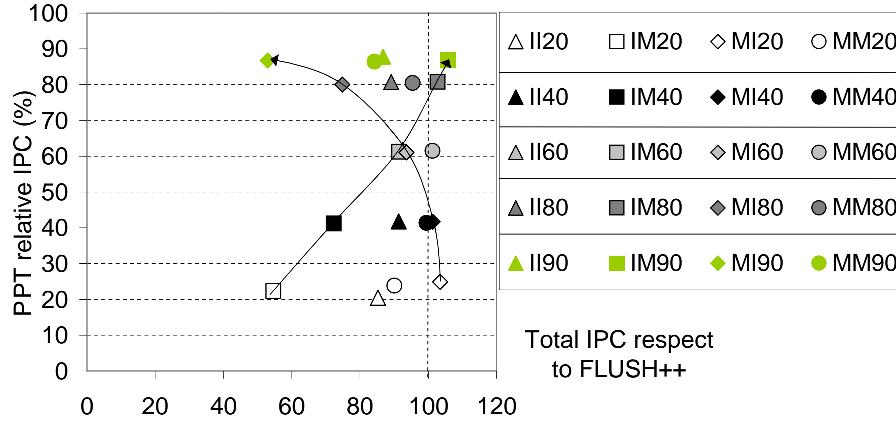


Fig. 6.9: QoS space where the y-axis shows the target percentage of full speed of the PPT, and the x-axis the throughput obtained by our mechanism with respect to *flush++*. The legend shows different workload types as well as target percentages required for the PPT.

take into account that this slowdow is also due to stop the UPTs during the sample phases.

- For the IM workloads (the squares in Figure 6.9), the UPT threads are MEM and experience many loads that miss in the L2 cache. Hence, these threads have low IPC and tend to occupy resources for a long time which has an adverse effect on the speed of the other threads in standard fetch policies. When the PPT is required to run at a low speed, the UPTs receive many resources and run at a speed that is close to the speed they would obtain under *flush++*. However, since the total throughput for *flush++* mainly derives from the speed of the PPT, total throughput is degraded. On the other hand, when we require a high relative IPC for the PPT which is ILP, total throughput increases because the PPT runs at a higher speed than it would under *flush++*. When the relative IPC of the PPT is 80% or higher, throughput is higher than with *flush++* because, although *flush++* is designed to deal with this situation by flushing a thread after an L2 miss, it also needs to re-fetch and re-issue all flushed instructions, degrading its performance.

- The MI workloads (the diamonds in Figure 6.9), present the opposite situation of the previous one. In this case, the total throughput in *flush++* comes largely from the UPT. When the PPT is required to run at a low speed, the UPTs get many resources and can run faster than they would under *flush++*. Hence, total throughput is higher than under *flush++*. When the relative IPC of the PPT is 40% or less, throughput is higher than with *flush++*. Conversely, when the PPT runs at a high relative IPC, it is allocated many shared resources to achieve this. As a result, total throughput drops since the full speed of the PPT is low, and also because the UPTs are denied many resources so that their speed becomes less than under *flush++*.

- For the MM workloads (the circles in Figure 6.9), neither the PPT nor the UPTs can make efficient use of resources. Hence, given that the PPT and the UPTs use resources in a similar way, there is not a significant variation in throughput when varying the target relative IPC of the PPT and it is almost the same as under *flush++*. Only for the extreme ends total throughput drops since in these cases resources are occupied by either PPT or UPTs when some of these resources would have been used better under *flush++*.

Summary

We can draw two main conclusions from these results. The first and most important one is that our QoS mechanism is capable of realizing a target IPC for a particular PPT within an error margin of less than 2% on average for relative IPCs from 10% to 80%. Another conclusion is that, at the same time, it maximizes total throughput, achieving relative IPCs of over 90% compared with the throughput obtained using *flush++*. In fact, *flush++* can be improved up to 40.2%, for the IM4 workload (`gzip` as the PPT, and `(mcf + equake + swim)` as the UPTs), when the PPT is required 90% of its full speed. The OS level job scheduler could take advantage of the trends shown in Figure 6.9 in order to improve throughput. For example, if the scheduler needs to deal with a workload consisting of a non-time critical MEM thread and a number of ILP threads, it can be advisable to run this MEM thread as an PPT with a low target percentage of its full IPC: the overall throughput can be larger than for *flush++*.

6.3.6 Results when there are two PPTs

In our experiments with two PPTs we vary the required target IPC for both High Priority Threads from 10% to 80% with a step of 10%. Hence, we study our mechanism for target IPCs for PPT0 and PPT1 of 10/10, 10/20, ..., 80/80, a total of 64 different combinations of target percentages. Furthermore, we consider combinations where the PPT0, the PPT1, and the UPTs can all be either ILP or MEM; this leads to 8 possible types of workload. We also study the situation when there are both 1 and 2 UPTs. Hence, there are $8 \times 2 = 16$ types and for each type 64 combinations of target percentages, hence $64 \times 16 = 1024$ experiments.

Table 6.4: Subset of workloads for 2-thread prioritization (condensed)

	PPT0	PPT1	UPTs
ILP	gzip	gcc	wupwise
	mesa	gap	apsi, crafty
MEM	twolf	twolf	mcf
	art	mcf	equake, swim

Table 6.5: Workloads for 2-thread prioritization

Thread type				3 threads	4threads
PPT0	PPT0	UPT	key	workload 0	workload 1
I	I	I	III	gzip, gcc, wupwise	gzip, gap, apsi, crafty
		M	IIM	gzip, gcc, mcf	gzip, gap, equake, swim
	M	I	IMI	gzip, twolf, wupwise	gzip, mcf, apsi, crafty
		M	IMM	gzip, twolf, mcf	gzip, mcf, equake, swim
M	I	I	MII	twolf, gcc, wupwise	twolf, gap, apsi, crafty
		M	MIM	twolf, gcc, mcf	twolf, gap, equake, swim
	M	I	MMI	twolf, twolf, wupwise	twolf, mcf, apsi, crafty
		M	MMM	twolf, twolf, mcf	twolf, mcf, equake, swim

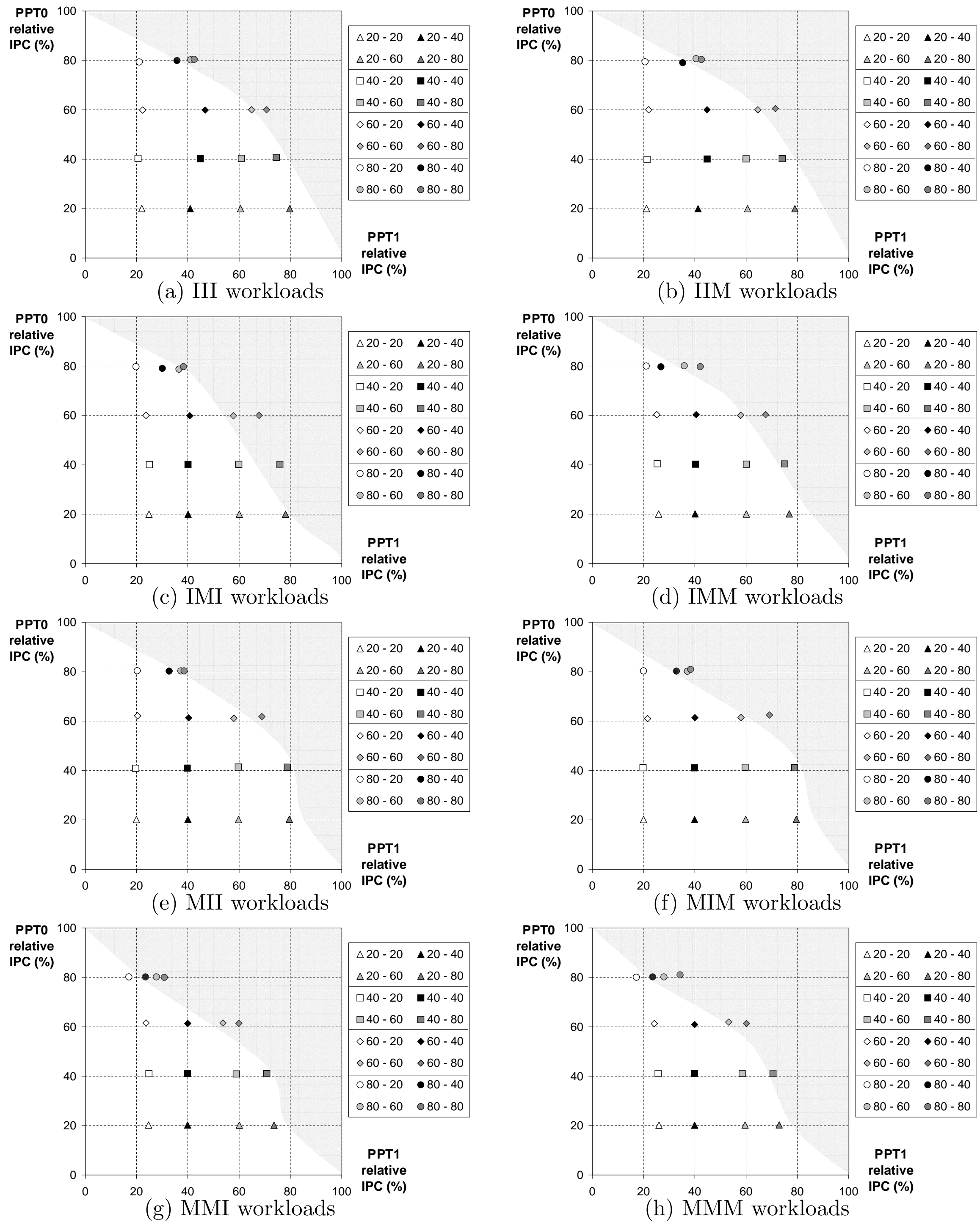
A complete study of all benchmarks is not feasible due to excessive simulation times. We have used the benchmarks shown in Table 6.4. For each type of workload, we use two different sets of threads to minimize any bias toward a specific set of threads. Hence, a total of $1024 \times 2 = 2048$ experiments have been carried out. Table 6.5 shows the composition of each of the eight types of workload. Since we want to have two workloads for each type, we also use the workloads obtained by replacing `gzip` and `twolf` as PPT0 by `mesa` and `art`, respectively. In this table the "key" column denotes the type of the workload, i.e., III means that the PPT0, the PPT1 and the UPT(s) are ILP.

Speed of PPT0 and PPT1

Figures 6.10(a) through 6.10(h) show the resulting QoS spaces for each type of workload. In these figures, the x-axis shows the relative IPC of PPT0 and the y-axis the relative IPC of PPT1. The legend shows the target percentage required for the PPT0 and for the PPT1, respectively. For clarity, we only show those percentages that are multiple of 20. The main conclusions we can draw from these charts are the following:

- In all cases we achieve the required percentage for the PPT0 or with an error lower than 2% on average. This indicates us that we are effectively isolating the execution of the PPT0 from the other threads. Furthermore, the addition of PPT1 does not affect PPT0.
- The are almost no differences when the type of the UPTs is changed. This indicates that in our mechanism the UPTs do not affect the execution of PPT0 and PPT1.
- For the PPT1 we do not achieve always the required percentage[5]. We differentiate 3 cases.

[5] In figures 6.10(a) through 6.10(b) the shadowed area represents, approximately, those points in which our mechanism is not able to accomplish with the OS requirements.



(a) III workloads
(b) IIM workloads
(c) IMI workloads
(d) IMM workloads
(e) MII workloads
(f) MIM workloads
(g) MMI workloads
(h) MMM workloads

Fig. 6.10: QoS space for all workload types

– When PPT0 is ILP (the IXX workloads), the combinations that fail are 40/80, 60/80, 80/40, 80/60, and 80/80.
– When PPT0 is MEM (the MXX workloads) the combinations that fail are 60/80, 80/40, 80/60, and 80/80.
– If both PPT0 and PPT1 are MEM (the MMX workloads), the combinations 60/60 and 40/80 also fail.

As a rule of thumb, we can realize the required percentages X and Y for PPT0 and PPT1, respectively, for all types of workload if $X + Y \leq 100$.

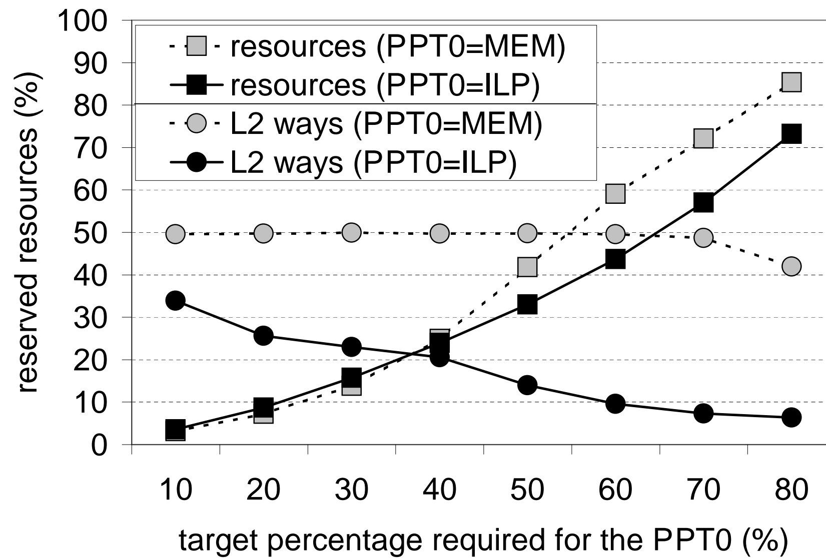


Fig. 6.11: Shared resources required to achieve a given relative IPC

That we fail in some cases to realize the target percentages for both PPT0 and PPT1 is not due to incorrectly isolating PPT1, but to an insufficient amount of resources inside the processor. Figure 6.11 shows the amount of shared resources (IQ entries and physical regs) and ways of the L2 cache required to achieve a given percentage of the relative IPC of the PPT0. Obviously, the higher the required percentage, the higher the amount that needs to be reserved. It is interesting to observe that when PPT0 is ILP and a relative percentage higher than 70% is required, then the amount of reserved resources is 50% or higher. This means that we cannot run both PPT0 and PPT1 at a relative IPC of 70% or higher. The same happens when the PPT is MEM when the required percentage is 60% or more.

Concerning the number of ways in the L2 cache, we observe that as we increase the required percentage of the full speed of PPT0, the amount of ways that need to be reserved decreases for both types of PPT0. This is because as we increase the percentage to achieve, the number of conflicts in the tune periods due to the other threads decreases because these threads are given less opportunities to execute.

For the MEM threads, 4 ways of the 8-way cache need to be reserved in order to avoid significant performance degradation and failing in achieving the given target percentage. Only when the target IPC for the PPT is 70% or more, less ways need to be reserved. This shows that when both threads are MEM the conflicts in the L2 cache can slowdown the PPT1 because it can only reserve up to 2 ways. For the ILP threads the number of reserved ways required are much lower and this number decreases as the target IPC increases.

Total throughput

In this section, we give a closer look at the throughput obtained by our mechanism for each of the different workload types and target percentages in turn. In all cases, two main observations characterize the obtained performance. First, better results are achieved if more resources are given to ILP threads than when these resources are devoted to MEM threads. Second, when we have to divide resources between threads with similar characteristics (all threads are MEM or ILP), then non-extreme target percentages lead to better performance results.

Figures 6.12(a) to 6.12(h) show for each workload type and required percentages for the PPT0 and the PPT1 the performance obtained with our policy with respect to the performance obtained with *flush++*. The results depending on the workload type are the following.

- Type III: given that all threads have similar behavior, the best results are achieved for intermediate percentages, that is, for percentages between 30 and 70%. The best result is achieved when the PPT0 requires 50% of its full speed and the PPT1 60%, obtaining 91% of the throughput obtained with *flush++*. This small drop in throughput is due to the fact that *flush++* is implemented on top of *icount*. *icount* achieves the good results for ILP threads and the division of resources disrupts its behavior. In some case, resources are reserved for threads that are stalled when these resources could have been used by the other threads. However, our mechanism obtains 82.4% of the performance obtained with *flush++* on average, showing that our mechanism does not need to pay too high a price in order to meet OS requirements.

- Type IIM: in this case, the higher the rIPC of both the PPT0 and the PPT1, the higher the throughput. This is caused by the fact that in these cases a lower number of resources is given to the UPTs that are MEM and that have a low IPC by themselves. Hence, under *flush++*, the UPTs tend to occupy resources for a long time which degrades the performance of the ILP threads. In our situation, the UPTs are not given the opportunity to hold many resources and as a result the PPTs reach a higher speed than under *flush++* and total throughput increases.

- Type IMI: better results are achieved as the rIPC of the PPT0 increases, excluding the extreme cases of 20/80 and 40/80. For a given rIPC of the

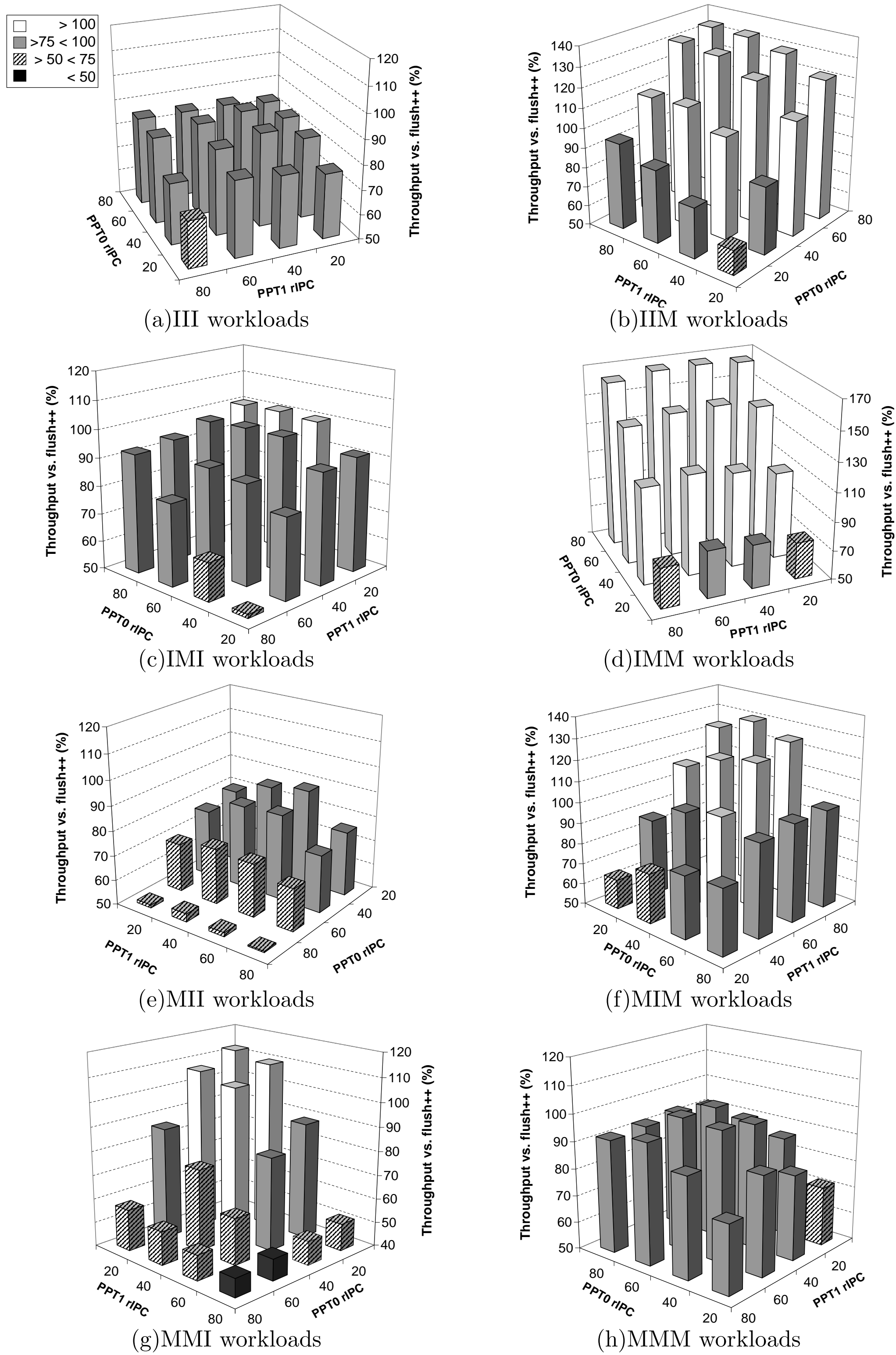


Fig. 6.12: Total throughput with respect to *flush++*

PPT0, better results are achieved when the rIPC of the PPT1 is small, because this PPT1 is MEM and then receives less resources. Since the UPTs are ILP, they obtain a high speed under *flush++*. Using our mechanism, they reach a much lower speed since many resources are dedicated to the PPTs and they cannot use these resources even if they would be idle. Hence, total throughput is degraded compared to *flush++* since the slow PPT1 is allocated many resources.

- Type IMM: the higher the rIPC of the PPT0, the better the results, since more resources are given to this ILP thread. For a given rIPC of the PPT0, better results are obtained for intermediate values of the PPT1. Using *flush++*, PPT0 would obtain a speed that is larger than 20% of its full speed. Hence, fixing the required speed of PPT0 at 20%, many resources go to slow MEM threads. This increases their throughput, but since this throughput is small to start with, total throughput is less than for *flush++*. Conversely, if PPT0 receives many resources in order to realize a high target percentage, it runs much faster than under *flush++*. Since the type of the workload is IMM, this implies that total throughput is much higher than for *flush++*.

- Type MII: the lower the rIPC of the PPT0, the better the results, since less resources are given to this MEM thread. For a given rIPC of the PPT0, better results are obtained for intermediate values of the PPT1. The worst results are obtained for high target percentages of PPT0. In these cases, only a small contribution to total throughput is obtained since this thread is MEM. Since the other threads are all ILP, they would run fast under *flush++* and in these cases they get too few resources to do so. Hence, total throughput is degraded.

- Type MIM: the higher the rIPC of the PPT1, the better the results, since more resources are given to the only ILP thread in this workload. Since in these cases, the ILP thread can really make progress, total throughput is increased with respect to *flush++* since under this fetch policy the ILP thread would experience much interference from the other MEM threads. For a given rIPC of the PPT1, better results are obtained for intermediate values of the PPT0.

- Type MMI: the lower the rIPC of both the PPT0 and the PPT1, the higher the throughput, because more resources are given to the UPTs that are ILP. As a result, the UPTs obtain a high speed which is higher than their speed under *flush++* and total throughput increases.

- Type MMM: as for the III workloads, all threads have similar behavior. Hence, intermediate target percentages for the PPT0 and the PPT1 lead to the best throughput results.

UPTs Throughput

Figure 6.13(a) shows the performance of the UPTs when they are ILP, that is, the average of the III, IMI, MII and MMI workloads (*XX*I). Analogously, Figure 6.13(b) shows the UPTs performance when they are MEM. In these figures, *rIPC* denotes the target percentage (or relative IPC) of a thread requested by the OS.

As expected, as the sum the target percentages becomes less, UPTs receive more resources and their IPC increases. Conversely, if this sum becomes larger, the throughput of the UPTs decreases. However, their throughput never becomes zero and even in the cases where a high target percentage for both PPT0 and PPT1 is to be reached, the UPTs still have some throughput. This shows that our mechanism reserves a minimal amount of resources for the PPTs to realize the required target throughput for them and can successfully use the remaining resources for the UPTs. Even when there is only a very limited number of fetch slots, IQ entries etc. that are not claimed by the PPTs.

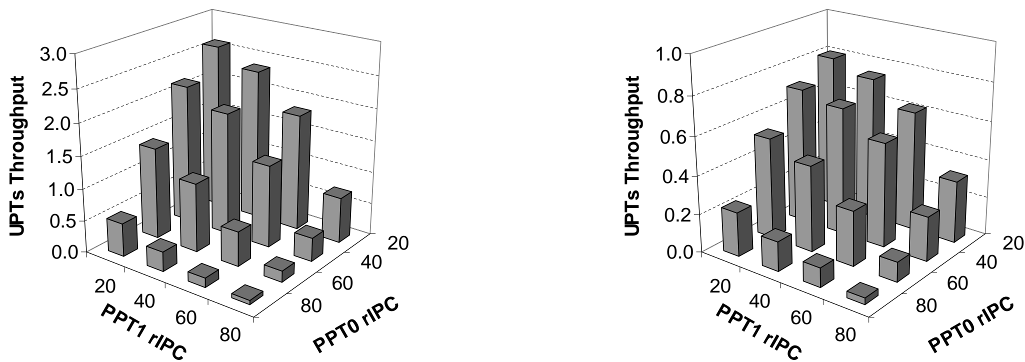


Fig. 6.13: UPTs throughput : (a)*XX*I workloads; (b)*XX*M workloads

Summary

The results have shown that our mechanism always achieves the required target IPC for the PPT0 with an error less than 2% on average. The addition of a minimum required IPC for a second thread, PPT1, does not affect the behavior of PPT0. For PPT1, the target is not achieved when the amount of resources is too small to realize both percentages at the same time.

Compared to a traditional superscalar processor and previous SMT fetch policies, our mechanism enables one degree of freedom more in scheduling job on an SMT processor: jobs can be given a certain *share* of the available resources. Our experiments show that a resource conscious job scheduler, by using this level of freedom, could significantly improve a traditional one on top of a fetch policy. On average, our mechanism achieves 90.4% of the performance achieved by *flush++*. For some workloads we achieve an improvement of 228%: for the IMM3 workload

(*mesa* as PPT0, *twolf* as PPT1 and *mcf* as UPT), *flush++* achieves an IPC of 1.782, whereas our mechanism, when the target percentage for the PPT0 and the PPT1 is 80%, leads to an IPC of 4.074.

6.3.7 Conclusions

Current Operating Systems view the different contexts of an SMT as multiple independent virtual processors. As a result, the OS schedules threads onto what it regards as processing units operating in parallel. However, in an SMT those threads are competing with each other for the resources of a single processor. On the other hand, in current SMT processors, resource allocation is done implicitly as determined by the fetch policy. Both factors lead to performance unpredictability, as the OS is not able to guarantee priorities or time constraints on the execution of a thread if that thread is to be run concurrently with other threads. As a result, we need to run time-critical applications in isolation, degrading overall performance.

We have proposed a novel strategy that enables a bidirectional interaction between the OS SMT processors, allowing the OS to run up to two time critical jobs at a predetermined IPC, regardless of the workload that these threads are executed in. As a consequence, this enables the OS to run time-critical jobs without dedicating all internal resources to them, and so, other low priority jobs can make significant progress as well.

We have tested our mechanism in a four-context SMT with up to two time-critical (Predictable Performance) threads (PPT0 and PPT1), for which we require a minimum IPC. When one thread is prioritized, our mechanism achieves for the entire range of workloads and target percentages, the required percentage or a little more. When two threads are to be prioritized at the same time, we always reach the target percentage for the PPT0 with an error less than 2% on average. The addition of a minimum required IPC for a second high priority thread, the PPT1, does not affect PPT0. For the PPT1, the target is not achieved only when the amount of resources is too small to accomplish this. This allows the OS, by designating 1 or 2 PPTs as well as their target percentages, to control the execution of these threads regardless of the workload they are executed in. As an additional advantage, in some cases our mechanism improves throughput comparing to *flush++*, achieving 90% of the performance of *flush++*. This shows that if the target were to maximize performance, then the synergy of the OS job scheduler and our workload and resource conscious policy could significantly outperform the best currently known fetch policies for SMT.

6.4 Low-Variability Performance

As shown in the introduction section of this chapter, a solution to improve the performance/cost ratio of processors is to allow threads to share hardware resources. SMTs share many hardware resources, and hence have a good performance/cost

ratio [41]. However, in SMT processors, threads may also interfere because they share many resources. This implies that the speed a thread obtains in one workload can differ significantly from the speed it has in another workload [14]. We refer to this by saying that an SMT processor has a high *variability*. Obviously, high variability is an undesirable property in a real time environment. In such environments, not only does the job scheduler need to take into account Worst Case Execution Times (WCETs) and deadlines when selecting a workload, but it should also know how the workload can affect the Worst Case Execution Time. Note that redefining WCET as the longest execution time in an arbitrary workload is not an option. By carefully selecting a workload, the WCET could be made arbitrarily large. Moreover, analytical approaches to WCET would fail miserably if they would need to take a context into consideration.

In the literature, several solutions have been proposed in order to improve this trade-off in SMT processors [15](this is the proposal we made in Section 6.3), [34][59]. The common characteristic of these solutions is that they assume knowledge of the average number of Instructions Per Cycle of applications when they are executed in isolation, called IPC_{alone} [6]. In other words, these solutions are *IPC based*. By comparing the IPC of a thread in a workload with its IPC_{alone}, these methods converge to a solution. In this section, we propose a different way of approaching the problem. Instead of using the IPC of applications to drive the solution, we use *resource allocation* that normally is implicitly driven by the instruction fetch policy. Our method makes explicit to the scheduler the amount of resources used by each thread. The scheduler adjusts this allocation to guarantee that applications meet their deadlines.

The advantage of our method is two-fold. First, it is well known that IPC values can be highly dependent on the input of an application. For some types of real-time applications, such as multimedia applications [33], this dependence is weak: the IPC of such an application is roughly independent from the input. But for other types of applications this is not the case. Our method does not require this information so that it is applicable to all types of real-time applications. Second, we achieve a similar or even better success rate than the approaches discussed above, while improving overall performance.

As far as we know, there does not exist a proposal that deals with this problem when the IPC_{alone} of applications is not known. In such a case, time-critical threads are given all the resources of the SMT [9]. This, of course, solves the problem but provides low throughput. In this section we propose a novel mechanism to enforce real-time constraints in an SMT based system. This mechanism consists of a small extension of the OS level job scheduler and an extension of the SMT hardware, called a *Resource Allocator*. Our approach is *resource based* instead of IPC based. By this, we mean that it relies on the amount of resources given to the time-critical thread. The job scheduler assembles a workload for the SMT processor and instructs the Resource Allocator to dedicate at least a certain amount of resources to the

[6] In this section we use the terms IPC_{alone} and full speed interchangeably.

time-critical thread so that it is guaranteed to meet its deadline. Apart from this, the Resource Allocator tries to adjust the resource allocation in order to maximize performance. The current section is focused on the Resource Allocator. In future work, we hope to give a working implementation of the job scheduler as well. Using our method, time-critical applications meet their deadline more than 98% of the cases considered while the non-critical applications obtain high throughput.

6.4.1 Background on real-time scheduling

In this section we give some background on real-time scheduling. In particular, we discuss some of the challenges to develop a real-time scheduler for SMT processors.

Real-time systems are characterized by a group of repetitive tasks, called a *task set*. For each task, the scheduler knows three main parameters.

- First, the *period*, that is, the interval at which new instances of a task are ready for execution.

- Second, the *deadline*, that is, the time before which an instance of the task must complete. For simplicity, the deadline is often set equal to the period. This means that a task has to be executed before the next instance of the same task arrives in the system.

- Third, the *Worst Case Execution Time* ($WCET$) is an upper bound on the time required to execute any instance of the task, which is guaranteed never to be exceeded.

In hard-real time systems, it should *a priori* be determined whether or not a task set is schedulable. In soft-real time systems, this requirement is not necessary since it is allowed that a small fraction of the deadlines is missed.

In soft-real time scheduling, many algorithms have been used to solve this problem in single-threaded systems (e.g., EDF or LLF). However, these algorithms are no longer sufficient in an SMT processor, since the execution time of a thread is unpredictable when this thread is scheduled with other threads. Algorithms should be adapted to meet this new situation.

As shown in [34], the problem of scheduling a set of tasks turns into two different problems in SMT systems. First, to select the set of tasks to run. This problem is called the *workload selection problem*. Second, to determine how resources are shared between threads. In this section, we focus on the latter problem that is also known in the literature as the *resource sharing problem*.

The high variability of SMTs implies that the task of a real-time job scheduler for SMT processors is much more complex and challenging than for single-threaded processors. When scheduling a job, the job scheduler must take into account the amount of resources given to a thread, which is implicitly decided by the instruction fetch policy in current systems, in order to ensure that it meets its deadline.

6.4.2 Experimental environment

In this section, we discuss both our baseline architecture used to run our experiments, the benchmarks we have used, and the metrics we employ to compare the different proposals.

SMT Simulator

We use an aggressive configuration, shown in Table 6.6: many shared resources (issue queues register, functional units, etc.), out-of-order execution, very wide superscalar, and a deep pipeline for high clock frequency. These features cause the performance of the processor to be very unstable, depending on the mix of threads. Thus, this configuration represents an unfavorable scenario where we evaluate our proposals. It is clear that, if those proposals work in this hard configuration, they will work better in narrower processors with fewer shared resources.

Table 6.6: Baseline configuration

Processor Configuration	
Default fetch policy	icount 2.8
Pipeline depth	12 stages
Fetch/Issue/Commit Width	8
Issue Queue Entries	64 int, 64 fp, 64 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	256 integer, 256 fp
(shared)ROB size	512 entries
Branch Predictor	16K entries gshare
Branch Target Buffer	256-entry, 4-way associative
Return Address Stack	256 entries
Memory Configuration	
Icache, Dcache	64 Kbytes, 2-way, 8-bank, 64-byte lines, 1 cycle access
L2 cache	2048 Kbytes, 8-way, 8-bank, 64-byte lines, 20 cycle access
Main memory latency	300 cycles
TLB miss penalty	160 cycles

Benchmarks

We use workloads consisting of two threads. The first thread is the *critical thread* (CT) that represents the thread with the most critical time restriction or the soft real time thread. The second thread is a *non-critical thread* (NCT) [7] that is assumed either to have less-critical time restrictions or to have no time restrictions at all. As critical threads we use programs from the MediaBench Benchmark suite, namely, *adpcm*, *epic*, *g721*, *gsm*, and *mpeg2*. We used both the coder and the decoder of these media applications. Hence, we use 10 media applications as critical threads. Table 2.2 shows the inputs for each of the MediaBench benchmarks.

[7] In this section we use the terms CT and PPT, and the terms NCT and UPT interchangeably.

We want to check the efficiency of the resource allocator under scenarios where the NCT requires many resources, and thus, where the performance of the CT could be more affected. For this reason, we use as non-critical threads benchmarks from the SPEC2000 integer and fp benchmark suite that require more resources than media applications. Each of the ten media applications is executed with 8 different benchmarks from the SPEC200 benchmark suite as non-critical thread. We have used *gzip*, *mesa*, *perlbmk*, *wupwise*, *mcf*, *twolf*, *art* and *swim*. These benchmarks were selected because they exhibit widely varying behavior. Some are memory bounded, which means that they generate many cache misses. Others are not but consume many computational resources. In our experiments below, we have used all pairs of media and general purpose applications, giving us a total of 80 workloads.

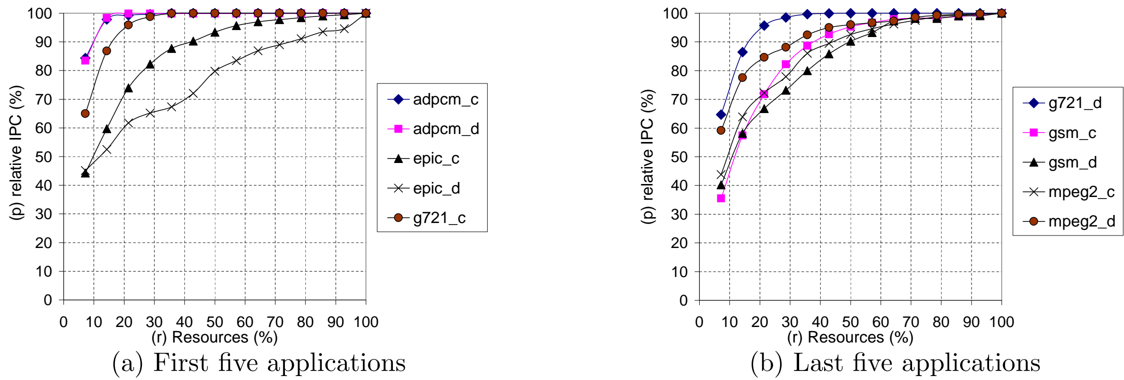
In order to check the efficiency of our Resource Allocator, we consider three different scenarios that differ in the stress that is put on our mechanism. The *worst-case utilization* U_w is defined as the fraction $U_w = \frac{WCET}{P}$ where $WCET$ is the Worst Case Execution Time and P is the period of an application. If the utilization is low, then $WCET$ is much smaller than the period and hence it should be relatively easy to guarantee deadlines. If, on the other hand, the utilization is high, then the critical thread must be given many or even all resources. In this case, it may happen more frequently that a CT misses a deadline.

In this paper, the WCET of an application is set equal to its real execution time when it is run in isolation in the SMT processor, called $ExecTime_i$. We consider three worst-case utilization factors called *low*, *medium*, and *high*. In the first case, we model a situation where the job scheduler has to schedule one task with a low worst-case utilization of 30%: we establish as a deadline for each task $3.3 \times ExecTime_i$. Hence, $U_i = \frac{WCET_i}{P_i} = \frac{ExceTime_i}{3.3 \times ExceTime_i} = 30\%$. In the second scenario, we model a medium utilization of 50% so that for each task its deadline is $2 \times ExecTime_i$. Finally, in the worst-case scenario, we use a high utilization of 80%. In this case, the deadline for each task is $1.25 \times ExecTime_i$.

Metrics

In all our experiments, we run the CT until completion. If the NCT finishes earlier, it is started again. When the CT finishes, we measure three values. First, the success rate (SR) that indicates the frequency the CT finishes before its deadline. In typical real-time systems, it is the responsibility of the OS level job scheduler to provide a high success rate. In our approach, this responsibility is shared between the job scheduler and the resource allocator. Second, we measure the performance of the non-critical thread. We want to give a *minimum* amount of resources to the critical thread to meet its deadline. The remaining resources are given to the non-critical thread in order to maximize its throughput.

Both these values are required to quantify the efficiency of our approach. For example, if a given CT has an utilization of 30% and the scheduler orders the resource allocator to assign to it 100% of the resources, the thread will obviously



(a) First five applications (b) Last five applications

Fig. 6.14: Relation between the amount of resources given to a task and its IPC.

meet its deadline. This provides a success rate of 100% but it does not provide high throughput nor does it show the efficiency of the resource allocator. Analogously, if a thread has an utilization of 90% and the scheduler orders the allocator to give it 10% of the resources, the thread likely misses its deadline and we do not know anything about the efficiency of the resource allocator.

As a third measure, in addition to the success rate, we measure the extra time required to finalize the CT for those cases in which the CT misses its deadline. Assume that the time required to execute the CT in a given workload, denoted by $Exectime_{CT}$, is larger than its deadline, $deadline_{CT}$, so that the CT misses its deadline. Then the *variance* is computed as: $variance_{CT} = \frac{(Exectime_{CT} - deadline_{CT})}{deadline_{CT}} \cdot 100\%$. For a given policy, we take the five cases in which the variance is highest and compute the average of these variances. We call this metric *Mean5WorstVariance*. If a policy has a success rate of 1, we have that $Exectime_{CT} \leq deadline_{CT}$ for each workload and in this case the variance is 0.

6.4.3 Dynamic resource partitioning

In this section, we discuss the extensions to the OS level job scheduler and the SMT hardware for implementing our scheme.

Overview of our approach

The basis of our mechanism is to *partition* the hardware resources between the critical and the non-critical thread and to reserve a *minimum* fraction of the resources for the CT that enables it to meet its deadline. In this way, we can also satisfy our second objective, namely, to increase as much as possible the IPC of the NCT. It is the responsibility of the job scheduler to provide the resource allocator with some information so that it can reserve this fraction for the critical thread.

When the WCET of a task is determined, it is assumed that this task has full access to all resources of the platform it should run on. However, when this task is executed in a multithreaded environment together with other tasks, it uses a certain

fraction of the resources. It is obvious that when the amount of resources given to a thread is reduced, its performance decreases as well. The relation between the amount of resources and performance is different for each program and may vary for different inputs of the same program. For the benchmarks used in this paper, we have plotted this relation in Figure 6.14. This figure shows the *relative IPC* [8] of each multi-media application when it is executed alone on the SMT as we vary the amount of window entries and physical registers given to it. This relative IPC is the percentage of the IPC the application achieves when executed alone on the machine and given all the resources, called IPC_{alone}. From this figure, we can see that if we dedicate 10% of the resources to the *epic* decoder, we obtain 50% of the speed it would have were it given the entire machine. Likewise, 10% of the resources dedicated to the *adpcm* decoder gives 95% of its IPC_{alone}.

Our proposed method exploits the relation between the amount of resources given to the critical thread and the performance it obtains. When the OS level job scheduler wants to execute a critical thread, given its $WCET$ and a period P, it simply computes the allowable performance slow down, S, given by $S = \frac{P}{WCET}$. For such a value of S, each instance of this job finishes before its deadline. Suppose the real execution time of this instance is T_i. Then, $T_i \leq WCET$. Hence, $S \cdot T_i = \frac{P}{WCET} \cdot T_i \leq \frac{P}{WCET} \cdot WCET = P$. Therefore, the value of S is a critical piece of information needed to establish a resource partitioning.

The following two issues need to be addressed. First, we need to determine which resources are being controlled by the resource allocator. Second, we need to decide whether the job scheduler or the resource allocator determines the exact amount of resources given to the critical thread. In the first case, a resource allocation is fixed for the entire period a critical thread is executing. We call this approach the *static* approach. In the second case, the resource allocator can dynamically vary the amount of resources dedicated to the critical thread. We call this approach the *dynamic* approach.

Resource allocator

The *resource allocator* controls the amount of resources that can be used by applications. It consists of a number of *resource usage* counters that track the amount of resources used by each application, one counter per resource. These counters are incremented each time a thread needs an additional instance of a resource and they are decremented each time an instruction releases an instance resource. For each thread in the SMT, there are also *limit* registers for each resource that contain the maximum number of instances the thread is allowed to use. These limit registers can be written by either the job scheduler in the static method or the resource allocator itself in the dynamic method. If an application tries to use more resources than it is assigned, its instruction fetch is stalled until resources are freed. We would like to emphasize that all added registers are special purpose registers. They do not belong

[8] The IPC is inverse to performance: $ExecutionTime = CycleTime \times \#Instructions \times (1/IPC)$.

to the register file. We discuss the hardware cost of our mechanisms in more detail in Section 6.4.5.

Resources

The first step in our approach is to determine the set of shared resources that has to be controlled to provide stability. In our architecture, the shared resources are the following: the fetch bandwidth, the issue queues, the issue bandwidth, the physical registers, the instruction cache, the L1 data cache, the unified L2 cache, and the TLBs. We have conducted a number of experiments to examine what the influence on variability is when we partially dedicate each of these resources to the CT.

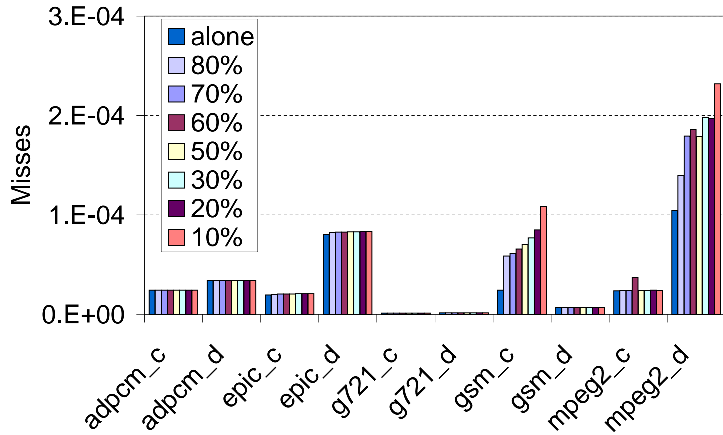
Caches and TLBs: Regarding TLBs, on average for all the experiments made in this section, the number of data TLB misses per instruction is 3.6×10^{-4} and the number of instruction TLB misses is 8.2×10^{-7}. Hence, the influence on the execution time of the CT is small. For this reason we do not control TLBs.

Regarding caches, we measured for all multi-media applications in all 80 workloads the average number of misses in each cache with respect to the number of committed instructions, as we vary the amount of resources given to it. The results are shown in Figure 6.15. We observe that there is an increase in cache miss rate caused by interference by another application in the workload. We observe as well that this interference is lower when the amount of resources given to the CT is higher and *vice versa*. This is caused by the fact that if the CT is allowed to use many resources, the NCT executes slower and hence uses caches less frequently and thus produces less interference.

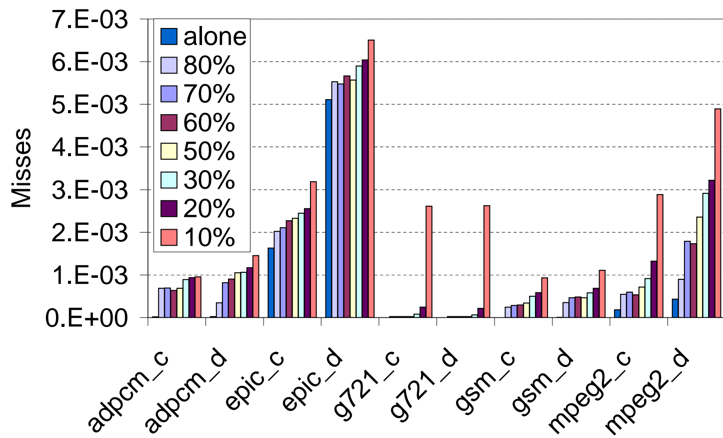
The absolute number of misses per committed instruction is low: lower than 2×10^{-4} for the instruction cache, 7×10^{-3} for the data cache, and 4×10^{-3} for the L2 cache. We can draw two conclusions from these figures. First, in the icache there is almost no interference between the CT and the NCT. Second, the interference introduced in the caches by a non-critical thread in a workload is so small that we expect that this only slightly affects the execution time of multimedia applications. As a result, we do not need to control how caches are shared between the threads.

Other resources: We systematically measured the effect of controlling resources other than the caches. We looked at the following resource partitions. *Nothing* means that we do not control any resource inside the SMT. Resources are implicitly shared as determined by the default fetch policy. *Fetch* means that we prioritize the CT when fetching instructions from the instruction cache. *Queues* and *Registers* mean that we give a fixed amount of entries of that resource to the CT. Furthermore, we made all combinations of these resources [9].

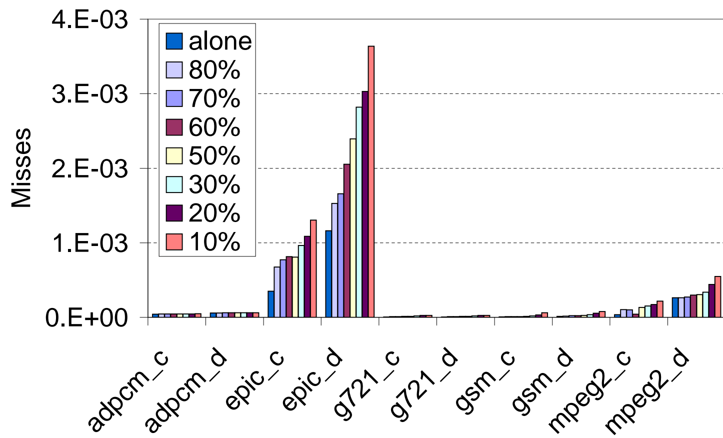
[9] The issue bandwidth provides small variations in the results. For this reason we do not show its results.



(a) instruction cache



(b) data cache



(c) L2 cache

Fig. 6.15: Cache interference introduced by the NCT as we vary the amount of resources given to the CT.

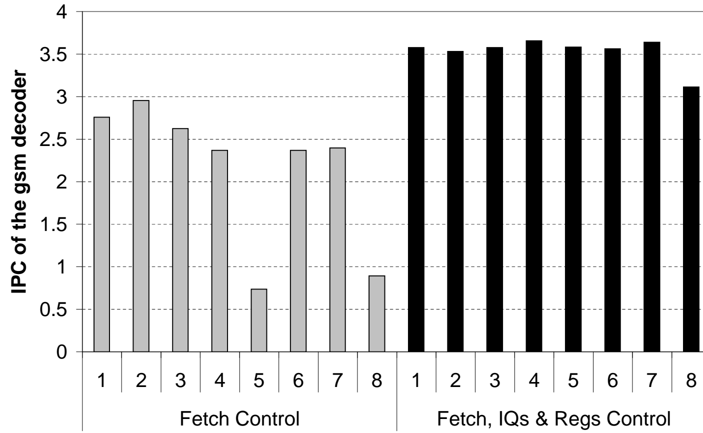
In Figure 6.16(a), we show for the *gsm* decoder benchmark its actual IPC values for two possible ways to partition resources: when we prioritize instruction fetch and when we partition the registers and the issue queues. From this figure, it is immediately clear that controlling the instruction fetch alone gives little control over the speed of the CT and the variability in IPC is large. On the other hand, controlling queues, registers and fetch does give much control over the speed of the CT and hence the variability is low.

In order to measure the sensitivity of the variability to resource partitioning more systematically, we proceed as follows. We used all pairs of media benchmarks as CT and spec2000 benchmarks as NCT. We measured execution times of the CT. For each CT from the MediaBench suite, we obtained 8 numbers, one for each spec benchmark. We computed the mean and the standard deviation of these numbers, and the fraction deviation/mean. In this way, we obtain a measure of the variability in execution time of a CT as we change the NCT, expressed relative to the average execution time. This allows us to average these values over all possible critical threads. This final average is the overall measure of variability used in this study. In Figure 6.16(b) we show these results. We can immediately observe that when we do not control resource allocation, we get a high variability of 40%. This can be interpreted as that in many cases the difference in execution time of a CT in an arbitrary context can be as high as 40% of the total execution time or even higher. If we only prioritize the instruction fetch of the CT, this variability is hardly reduced. The most important resources to control are the registers and the issue queue entries. The best results are obtained when we control everything: we give the CT priority in the fetch stage and reserve a certain amount of registers and issue queue entries for it. These are the resources controlled by our Resource Allocator from now on.

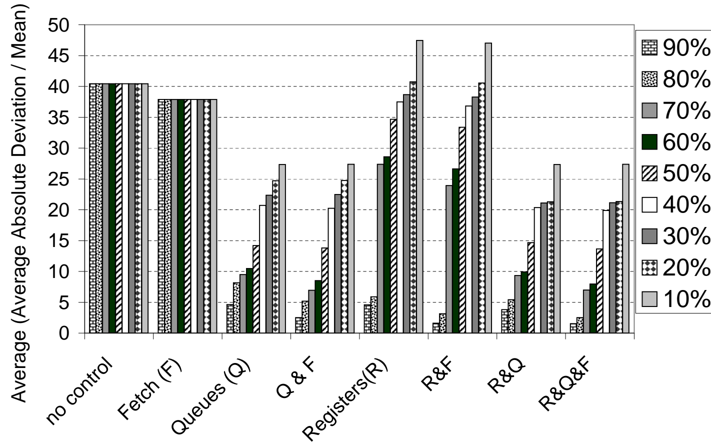
Regarding the percentage of resources given to the CT, we show that as we decrease this amount the variability increases. For low percentages (10% or 20%) variability is even higher than when no control is carried out. This is mainly because when the CT uses few resources, the NCT executes more instructions, causing more interferences. In addition, every time the CT misses in the cache, it has not enough resources to hide this latency, even for L1 data cache misses. Hence, we conclude that if we give less than 20% of resources to the CT, its deadline could be compromised.

Static approach

In this section, we discuss our static approach to resource partitioning. In this approach, the job scheduler computes *a priori* the resource partitioning that is used throughout the entire period of the critical thread. In Figure 6.14, we have plotted the relation between the amount of resources dedicated to the CT and the performance it obtains. It clearly follows that, for all benchmarks considered in this paper, the relation between performance and amount of dedicated resources is super-linear. That is, if we dedicate $X\%$ of the resources, we obtain more than $X\%$



(a) IPC when we control the fetch (gray bars), and the fetch, the IQs and regs. (black bars).



(b) Average of the fraction (Standard Deviation / Mean).

Fig. 6.16: Variability in CT's IPC in different workloads as we vary the resources under control.

of IPC_{alone} and in some cases much more. Since the job scheduler knows the WCET of the critical thread and the period, P, in which it should execute, it knows the slow down the CT can suffer: the slow down factor $S = \frac{P}{WCET}$ discussed above. Hence, given this fraction S of the performance of the CT, it needs to compute a function $f(1/S) = Y$ to determine that the CT needs $Y\%$ of the resources to obtain this performance. We call such a function a *performance/resource function* or *p/r function*. These p/r functions can be considered to be approximations to the inverse of the curves shown in Figure 6.14. Hence, when the job scheduler assembles a workload with a certain application as critical thread, it computes the value of S and determines the corresponding value $Y = f(1/S)$ for a p/r function f. Then it instructs the resource allocator to reserve $Y\%$ of the resources for this critical thread. In the next section, we discuss performance/resource functions in more detail.

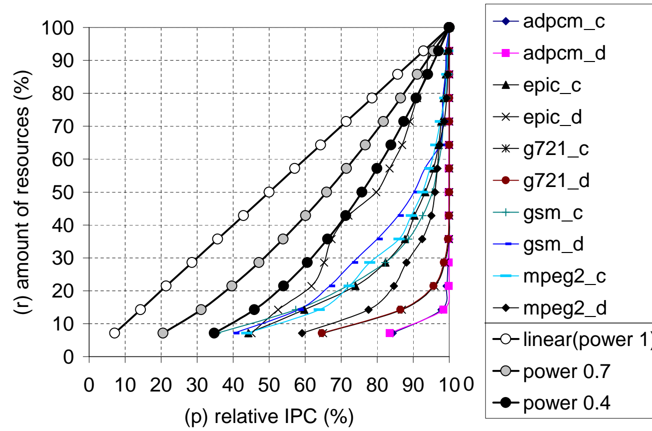
Performance/resource functions: Figure 6.17(a) shows the actual p/r relation of each thread and several p/r functions that approximate this actual p/r relation, which can be used by the job scheduler. These functions are plotted as circles in the figure. We show several functions that are given by $r = f(p) = p^{1/value}$ for *value* equal to 1 (linear), 0.7, and 0.4. For lower *values*, the amount of resources given to the CT is reduced and the actual p/r relation is better approximated. This may be positive since we allow the NCT to use more resources. However, this may also compromise the success rate.

For our experiments discussed in the next section, we use the p/r functions described above. We moreover use another p/r function that is more directly based on the graphs shown in Figure 6.14, called the *empirical* function. In order to construct this function, we empirically determine for each Multimedia Benchmark and for each possible value of p, the value of r that leads to a good success rate and achieves high NCT performance. We may consider this function as the upper bound of our static method. Figure 6.17(b) shows an example of how the function *empirical* is determined. The diamonds show the actual p/r relation for the gsm_d benchmark. The circles show the approximation we used. Note that this approximation is slightly larger than the corresponding value in the curve in Figure 6.14 in order to take into account interference by the NCT, mainly for low percentages of p.

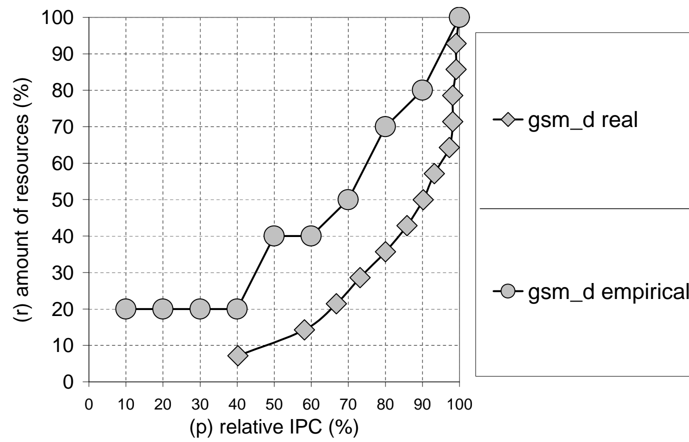
Dynamic approach

In this approach, the resource allocator dynamically determines the amount of resources given to the critical thread. The main advantage of this method is that it adapts to program execution phases, increasing overall performance. In the next section, we show that the dynamic approach provides better results than the static approach but it is only applicable if the application under consideration has a number of characteristics that we discuss in more detail below. The main advantage of the static approach is that it can be used always.

The mechanism is based on the observation that in order to realize $X\%$ of the overall IPC for a given job, it is sufficient to realize $X\%$ of the maximum possible IPC at every instant throughout its execution. In the present case, if we want to slow down an task with a factor S, it is sufficient to slow it down with a factor S at every instant. The dynamic approach that exploits this observation is a simplification of the method we proposed in [15][16] (see Section 6.3), called *Predictable Performance* or *PP*. The resource allocator distinguishes two phases that are executed in alternate fashion. We briefly describe these phases below. For more information, please consult [15]. During the first phase, the *sample phase*, all resources under control are given to the CT and the NCT is temporarily stopped. As a result, we obtain an estimate of the current IPC_{alone} of the CT which we call the *local* IPC_{alone}. The sample phase starts with a warm up period of 50,000 cycles to remove pollution by the NCT from the shared resources. Next, we measure the IPC_{alone} of the critical thread in an *actual-sample* phase of 10,000 cycles.



(a) Different arguments for the Power function



(b) Adhoc function

Fig. 6.17: Different performance/resources functions

During the second phase, the *tune phase*, the NCT is allowed to run as well. Our mechanism dynamically varies the amount of resources given to the CT to achieve an IPC that is equal to the local $IPC_{alone} \times 1/S$. The tune phase, which lasts 300,000 cycles, is divided in periods of 15,000 cycles during which the realized IPC of the CT is measured. If this measured value is lower than required, the CT is assigned more resources. If it is higher than required, resources are taken away from the CT and given to the NCT.

The main difference between the present dynamic approach and the *Predictable Performance* mechanism from [15][16] is that in *PP* the real value of IPC_{alone} of the CT is used to compute resource allocations in the tune phase. This value has to be provided by the OS, in contrast to the present approach that does not require this value. Moreover, the present approach controls fewer resources. In particular, we do not exercise control over the caches, in contrast to *PP* in which L2 cache miss

rates of the critical thread are monitored and this information is used to dedicate part of the L2 cache exclusively to the critical thread.

The main difficulty in our dynamic method is to measure accurately the local IPC_{alone} of the CT, due to the pollution created by the NCT in the shared resources. As we have shown in [15][16], the main source of interaction among the CT and the NCT is the L2 cache. This pollution stays for a long time, up to 1.5 million cycles. We have analyzed the pollution caused by the NCT in this resource in detail. We found that for multi-media applications the measured value of the IPC_{alone} is 1% lower than the real value. For SPEC benchmarks used in [15][16], it is 8% lower. The main reason for this is that media applications have a smaller working set than spec benchmarks. For this reason, an NCT does not interfere as much with media applications as with spec benchmarks. As a result, we can use a more simple resource partitioning algorithm for media applications than the algorithm from [15][16] that is geared toward general purpose applications.

We conclude that, if applications under consideration have a small working set in comparison with the L2 cache size, then they are unlikely to be affected by a NCT with a much larger working set. As a result, the IPC measured in the sample phase is closer to the actual IPC_{alone}, what allows us to leave the L2 miss rate out of consideration, thereby considerably simplifying the mechanism. If this condition is not satisfied, the dynamic approach cannot be applied and we have to resort to either the static approach discussed above or to the expensive *PP* mechanism described in [15][16].

To summarize, in the dynamic approach, the job scheduler provides the value $1/S$ to the resource allocator. Next, the resource allocator determines the IPC_{alone} of that instance of the task during a sample phase and reduces its IPC by a factor of $1/S$ during the subsequent tune phases. This implies that the CT can meet its deadline and that we minimize the amount of resources given to the CT, enabling high performance of the NCT.

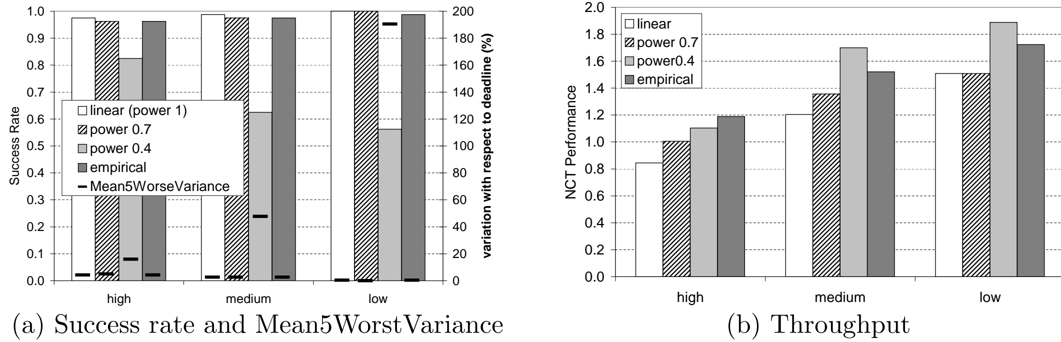
6.4.4 Simulation results

In this section, we present the results of the static and dynamic approaches. Moreover, we show the results obtained using the *Predictable Performance* mechanism we presented in [15] (Section 6.3) and a fetch control like mechanism based on [9][59].

Static method

Figure 6.18 shows the success rate and the performance for the different p/r functions used in the static method. In figure 6.18(a), bars show the success rate and are measured on the left y-axis. Lines show the Mean5WorstVariance and are measured on the right y-axis.

In figure 6.18(a), we can see that the linear p/r function provides the best success rate. We also observe that when the p/r function is more aggressive, the success rate decreases. This is intuitively clear, since we reduce the amount of resources given to



(a) Success rate and Mean5WorstVariance (b) Throughput

Fig. 6.18: Success Rate and throughput as we change the p/r relation

the CT. All functions, except the function $f(p) = p^{1/0.4}$, achieve a good success rate and Mean5WorstVariance. As we move from high to low utilization scenarios, the success rate improves. On average, the success rate is 0.987, 0.979, and 0.671 for the linear, $f(p) = p^{1/0.7}$, and $f(p) = p^{1/0.4}$ functions, respectively. For the empirical function the success rate is 0.975. The Mean5WorstVariance is 1%, 2.6%, and 87% and 2.4%, respectively.

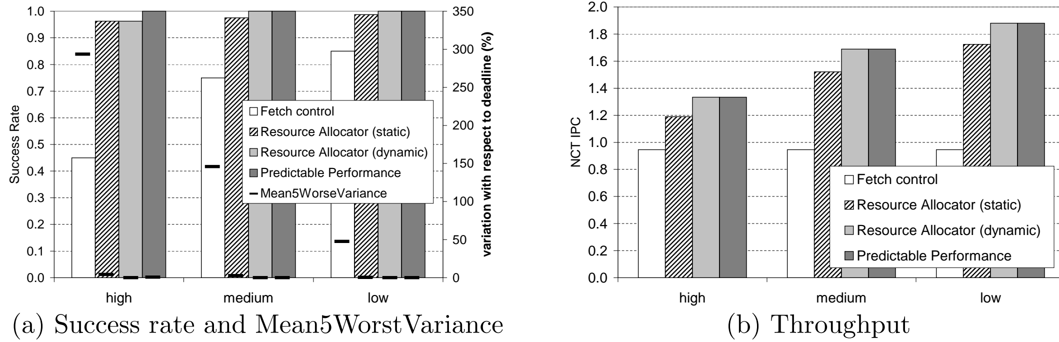
Figure 6.18(b) shows the average IPC of the NCT, averaged over all experiments. We see that the function $f(p) = p^{1/0.4}$ achieves the best performance results. However, this is at the cost of success rate. Hence, we conclude that this function is too aggressive. We observe that as we increase the aggressiveness of the p/r function, we obtain more performance for the NCT. This is because more aggressive p/r functions provide the CT with fewer resources.

We conclude that the *empirical* performance/resource function performs best. If this function is too difficult to obtain in some circumstances, the p/r function $f(p) = p^{1/0.7}$ performs only slightly worse. Recall that a p/r function is computed by the OS level job scheduler and hence is implemented in software. Hence we can easily use complex functions like the functions discussed above.

Dynamic method

In this section, we present the results of the dynamic method and moreover compare this mechanism with previous approaches. For this experiment, we have also considered the Predictable Performance mechanism proposed in [15] and a prioritization-aware fetch policy [9][59] which we call *fetch control* in this section. This mechanism always prioritizes the CT when fetching instructions. We also compare our results with the *adhoc* p/r function for the static method.

Figure 6.19(a) shows the success rate for the different approaches. If we just control fetch, we see that we obtain a low success rate and a high Mean5WorstVariation, even for the low utilization scenario. The predictable performance approach obtains a success rate of 1, and hence a Mean5WorstVariation of 0. This is mainly due to the fact that this policy uses knowledge of the IPC_{alone} of each CT that allows it to compute dynamically how far the current IPC of the CT is from the target IPC.



(a) Success rate and Mean5WorstVariance (b) Throughput

Fig. 6.19: Success Rate and throughput of our approach and previous approaches

In this way, this mechanism converges to the target IPC. Our mechanism does not require this information but achieves a success rate of 0.9875 nevertheless. It also has a low Mean5WorstVariance of 1.63%. In the static method using the *adhoc* p/r function, the success rate is 0.975 and the Mean5WorstVariance is 2.4%.

Regarding throughput, we can see that our dynamic method achieves the same performance as Predictable Performance. Our static method achieves 9% less performance, but still much more performance than the fetch control method, up to 56% more in the low utilization scenario.

6.4.5 Implementation

The two proposals shown in this section require some hardware to control the amount of resources given to the CT and NCT. In this section, we present the hardware changes in our baseline architecture to provide such functionality. Finally, we show how the OS and the hardware collaborate in order to deal with time requirements.

Hardware to control resource allocation

The objective of this hardware is to ensure that the CT is allowed to use at least a given amount of each shared resource. The tasks done by this hardware are as follows: track, compare, and stall.

Track: We need a *resource usage counter* for each resource under control, both for the CT and the NCT. Each counter tracks the number of slots that each thread has of that resource. Figure 6.20 shows the counters required for a 2-context SMT if we track the physical registers. Resource usage counters are incremented in the decode stage (indicated by *(1)* in Figure 6.20). Register usage counters are decremented when the instruction commits (2). We also control the occupancy of the IQs. Hence, 3 limit resources and 3 queue usage counters are required, one for each queue. Queue usage counters are decremented when instructions are issued from the issue queues. All added registers are special purpose registers. They do not belong to the register file. The design of the register file is left unchanged with respect to

the baseline architecture. The implementation cost of these counters depends on the particular architecture. However, we believe that it is low due to the fact that current processors have tens of performance and event counters e.g, the Intel Pentium4 has more than 60 performance and event counters [5][60].

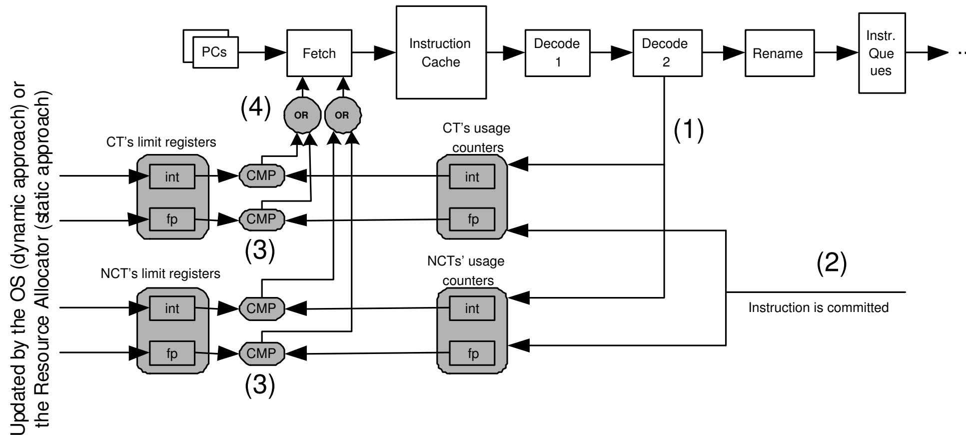


Fig. 6.20: Hardware required to implement our mechanism

Compare: We also need two registers, *limit registers*, that contain the maximum number of entries that the CT and NCT are entitled to use. These registers are modified by the OS periodically as we discuss next. In the example shown in Figure 6.20, we need 4 counters: one for the fp registers and one for the integer registers for both the CT and NCT. Every cycle we compare the resource usage counters of each thread with the limit registers (3). If a threads is using more slots than given to it, then a signal is sent to the fetch stage to stall instructions fetch from this thread (4).

Stall: If this signal is activated the fetch mechanism does not fetch instruction from that thread until the number of entries used for this thread decreases. Otherwise, the thread is allowed to compete for the fetch bandwidth as determined by the instruction fetch policy.

OS/hardware collaboration

For the static approach, the OS only has to update the values of the (special purpose) limit registers in order to accomplish with tasks' deadlines. When the OS schedules a task for execution, it also sets the value of these registers [10].

For the dynamic approach, the OS sets the percentage of the IPC_{alone} of the CT that the hardware has to achieve. This requires the addition of one register. In

[10]Note that, if the limit registers of the CT and the NCT are set to the maximum number of resources, no thread is stalled. That is, we would have a standard SMT processor guided by the fetch policy.

addition to the hardware discussed in the previous section, we use a Finite State Machine (FSM) to dynamically change the resource allocation to converge to the target IPC. This FSM is the same we presented in Section 6.3.3 (Table 6.1).

6.4.6 Conclusions

In this section, we have proposed two novel approaches to the problem of enabling SMT processors for soft-real time systems, namely, LVP static and LVP dynamic. The main problem of using SMT processors in real-time systems is that in an SMT processor threads share almost all hardware resources. This may cause interference between threads which, implies that the speed a thread obtains in one workload can be very different from the speed it has in another workload [14]. In contrast to previous approaches, our methods do not require any knowledge beyond information that is traditionally used by the OS level job scheduler, namely, Worst Case Execution Time and the Period of the time-critical thread. Nor do our methods require extensive profiling of candidate workloads like some other methods do [34][59]. Our methods are based on resource partitioning, in contrast to previous approaches that are IPC based, reserving a minimum fraction of all resources for the critical thread so that it can just reach its deadline. In this way, the non-critical threads also receive as many resources as possible so that their throughput is maximized at the same time. In the first method, the job scheduler determines the fraction of resources dedicated to the critical thread and this fraction is fixed during the entire period. In the second method, the SMT hardware extension of a resource allocator dynamically adjusts the amount of resources for the critical thread, thereby adapting to program phases which increases the throughput of the non-critical threads even more. We have compared our approaches to two previously published mechanisms, namely, *fetch control* [9][59] and *Predictable Performance* [15] (Section 6.3). We have shown that we significantly outperform *fetch control* and are almost as good as *Predictable Performance* using a less complex mechanism. On average, the critical thread meets its deadline in 98% of the cases considered.

6.5 Chapter summary

In this chapter, we have proposed two different mechanisms that enable the use of SMTs in real-time systems. Both approaches are based on the idea of explicit resource allocation. Next, we discuss some issues related with these mechanisms, like their cost and their applicability.

Concerning the cost of the different mechanisms, the *fetch control* mechanism used in [59] only prioritizes the fetch of the critical thread and hence has the lowest cost. Our static mechanism (Section 6.4) needs to keep track of how many resources are used by each thread and hence is more expensive. However, the cost for doing this is not high, as shown in the implementation section. Since the required percentage that the critical thread should receive is provided by the job scheduler, the base

Table 6.7: Comparing approaches shown in this chapter

Metric	fetch control	static	dynamic	PP
Success rate	-- (<75%)	++ (>95%)	++	++
Throughput NCT	--	+	++	++
Applicability	++	++	+	-
Cost	--	--	-	+

resource allocator is sufficient to implement our static method. Our dynamic method (Section 6.4) is more complex. Apart from the resource allocator that is required to monitor that threads do not exceed their share of the resources, a mechanism in hardware is required to sample the IPC_{alone} of the critical thread during the sample phase and to periodically determine the IPC of this thread during the tune phase. Moreover, logic is required to suspend the NCT during the sample phase and to adjust resource allocation during the tune phase. Finally, Predictable Performance (Section 6.3) requires all this plus extra logic to monitor the L2 cache. Moreover, the logic required to determine resource allocation after the sample phase is more complex than the logic required by the dynamic mechanism.

Concerning the applicability of the various approaches, both *fetch control* and our static method can be used for all applications. Our dynamic method requires that the non-critical thread does not interfere too much with the critical thread in the L2 cache. Predictable Performance requires that all instances of an application have more or less the same IPC. Fortunately, it has been shown [33] that media applications generally have these properties so that it can be applied to media applications. We can summarize all aspects in Table 6.7. In this table, we use the following symbols in order to indicate the suitability of each approach on the aspect considered: ++ (very high), + (high), - (low), and -- (very low).

Depending on the properties of applications that need to run on the system, the amount of hardware available to provide soft real time functionality, and the required success rate, a designer of an embedded, real-time system can choose one of the alternatives discussed in this chapter. If there is hardly any room to implement a real time mechanism, *fetch control* can be used; this has a poor success rate but costs next to nothing. If there is a modest amount real estate available and the success rate must be reasonably high, our static method is best suited. If, on the other hand, the success rate must be 1 then Predictable Performance can be used, at the cost of a complex implementation. In situations in between, our dynamic method may be a good candidate.

7

Conclusions

CHAPTER 7

CONCLUSIONS

This chapter lists the main conclusions of this thesis as well as future directions.

7.1 Thesis conclusions

SMTs are being used in an increasing number of systems that range from high-performance systems [36][47] to real-time systems [10][41].

The main advantage of using SMT processors is that threads running on an SMT share many more resources than in other types of processors. This improves the performance/cost/consumption ratio of SMTs. However, the main disadvantages of existing SMTs are two-fold:

- In the collaboration between the OS and the SMT, the OS perceives the different contexts of an SMT as multiple, independent processing units operating in parallel. However, these virtual processors are not truly independent since threads scheduled at any given time compete with each other for the resources of a single processor.

- Current SMTs are designed with the main objective of increased throughput, lacking flexibility in providing other objectives. A fetch policy decides how internal processor resources are allocated to the threads.

The main contribution of this thesis is that we have proposed, for the first time, the concept of *explicit resource allocation* in SMT processors as a way to solve these two problems. The explicit reosurce allocation enables Quality of Service in SMT processors. We demonstrated what hardware support is necessary to provide explicit resource allocation. We also showed how explicit resource allocation improves the efficiency of SMTs while enabling the use of SMTs in a wider range of systems.

- For high-performance systems we proposed the flush++ and dwarn fetch policies, that improve previous policies in both throughput and fairness. At the same time, these two fetch policies reduce the number of instructions flushed from the pipeline. These two policies are the precursors of the concept of explicit resource allocation.

  - Flush++: it improves the flush [63] policy in throughput and fairness while reducing the number of instruction flushed from the pipeline due to L2 data cache misses.
  - DWarn is a simple mechanism that provides good results while erasing the need of flushing instructions from the pipeline.

  Next, using the concept of explicit resource allocation we propose a resource allocation policy, called dcra. Dcra moves one step further than existing instruction fetch policies: it uses a direct resource allocation policy, instead of fully relying on the fetch policy to determine how critical resources are shared between threads. Dcra identifies those threads competing for a given resource and those threads that should be able to give part of their resources to other threads without damaging performance. Dcra continuously distributes resources taking these classifications into account and directly ensures that no resource-hungry thread exceeds its allocation. Dcra improves all known instruction fetch policies.

- For real-time systems, we proposed a mechanism, called predictable performance (PP), that allows the OS to execute up to two time-critical jobs at a predetermined IPC, regardless of the workload that these threads are executed in.

  - When 1 thread is prioritized, our mechanism achieves for the entire range of workloads and target percentages, up to 80%, the required percentage or a little more.
  - When 2 threads are to be prioritized at the same time, we always reach the target percentage for the first Predictable Performance Thread or PPT0 with an error less than 2% on average. The addition of a minimum required IPC for a second predictable performance thread, the PPT1, does not affect PPT0. For the PPT1, the target is not achieved only when the amount of resources is too small to accomplish this.

  This enables the OS to run time-critical jobs without dedicating all internal resources to them, and hence, other low priority jobs can make significant progress as well. As an additional advantage, in some cases, PP improves throughput compared to *flush++*, achieving 90% of the throughput of the *flush++* fetch policy.

  Next, we propose two resource based mechanisms with the same objective. These mechanisms provide results similar to PP and, in addition, are suitable for a wider range of applications. In contrast to previous approaches, including PP, these methods do not require any knowledge beyond information that is traditionally used by the OS level job scheduler, namely, Worst Case Execution Time and the Period of the time-critical thread. Nor do our

methods require extensive profiling of candidate workloads like some other methods do [34][59]. Our methods are based on resource partitioning, instead of previous approaches that are IPC based, reserving a minimum fraction of all resources for the critical thread so that it can just reach its deadline.

We discussed the pros and cons of all both mechanisms to explore the design space of real time enabled SMT processors in detail.

To sum up, all the results presented in this thesis show that explicit resource allocation provides Quality of Service for SMT processors.

7.2 Future work

This thesis opens up several topics from which we emphasize the following:

- High-performance systems: we saw that the dcra mechanism could be improved for MEM workloads where the pressure on resources is high. We are now working in an hybrid mechanism, called dcra-flush++. The objective is to improve the performance of dcra while reducing as much as possible the number of flushed instructions.

- Real-time systems: we analyzed the design space for the hardware support required to successfully use an SMT processor in a real-time system. The next natural step is to propose a real-time job scheduler that uses this hardware support in order to improve system performance.

We are already working on some of these topics. We hope to deal with the other topics in the near future.

Publications

Conferences

- Dynamically Controlled Resource Allocation in SMT Processors. Francisco J. Cazorla, Enrique Fernandez, Alex Ramirez and Mateo Valero. The 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Portland, USA. December 2004.

- Predictable Performance in SMT Processors. Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez and Mateo Valero. ACM Conference on Computing Frontiers (CF-2004). Ischia, Italy. April 2004.

- DCache Warn: An I-Fetch Policy To Increase SMT Efficiency. Francisco J. Cazorla, Enrique Fernandez, Alex Ramirez and Mateo Valero. International Parallel and Distributed Processing Symposium (IPDPS 2004). Santa Fe, New Mexico. April 2004.

- Improving Memory Latency Aware Fetch Policies for SMT Processors . Francisco J. Cazorla, Enrique Fernandez, Alex Ramirez and Mateo Valero. International Symposium on High Performance Computing (ISHPC-V). Tokyo, October 2003. Best student paper award. Published in Lecture Notes in Computer Science (LNCS) 2858.

- Feasibility of QoS for SMT by Resource Allocation. Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez and Mateo Valero. EURO-PAR 2004. Pisa, Italy. September 2004. Published in Lecture Notes in Computer Science (LNCS) 3149.

- Enabling SMT for Real-Time Embedded Systems. Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez and Mateo Valero. 12th European Signal Processing Conference (EUSIPCO). Vienna, Austria. September 2004.

- Approaching a Smart Sharing of Resources in SMT Processors. Francisco J. Cazorla, Enrique Fernandez, Alex Ramirez and Mateo Valero. Workshop on Complexity-Effective Design (WCED). Munich, Germany. June 2004.

- Implicit vs. Explicit Resource Allocation in SMT Processors. Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez and Mateo Valero. EUROMICRO Symposium on Digital System Design. 2004. Invited Paper.

- Architectural Support for Real-Time Task Scheduling in SMT Processors. Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez and Mateo Valero. International conference on compilers, architectures and systems for embedded systems (CASES), San Francisco, USA. September 2005.

Journals

- QoS for High Performance SMT Processors for Embedded Systems. Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez and Mateo Valero. IEEEMicro Special Issue on Embedded Systems: Architecture, Design, and Tools. July/August 2004.

- Improving Memory Latency Aware Fetch Policies for SMT Processors. Francisco J. Cazorla, Enrique Fernandez, Alex Ramirez and Mateo Valero. The International Journal of High Performance Computing and Networking. Special issue on ISHPC-V. No.2. April, 2004.

Submitted Papers

- Predictable Performance in SMT Processors: Synergy Between the OS and SMTs. Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez and Mateo Valero. IEEE Transactions on Computers.

List of Figures

1.1 A possible classification of multithreaded architectures 4

2.1 Block Diagram of our baseline architecture 12
2.2 Stages in our baseline architecture 13

3.1 Schema of the related policies 18
3.2 l2mp mechanism 23
3.3 Effectiveness of the pattern predictor and the 2bc 25
3.4 Comparing current policies 26
3.5 Undetected L2 misses when the pattern predictor is used 27
3.6 Improvement of the l2mp+ policy over the l2mp policy 27
3.7 Timing of the flush policy 28
3.8 Timing of the improved flush policy 28
3.9 Improvement of the flush+ policy over the flush policy. 29
3.10 Improvement of the stall+ policy over the stall policy. 30
3.11 Timing when a thread with high L2 miss rate is executed 30
3.12 Improvement of the flush++ policy over the flush, stall, flush+ and stall+ policies. 31
3.13 EF decrement of flush++ with respect to flush and flush+ 32
3.14 dwarn fetch bandwidth utilization 37
3.15 Dwarn resource utilization 38
3.16 Throughput results 40
3.17 Extra Fetch caused by the flush policy 41
3.18 Hmean improvement of dwarn over the other policies 43
3.19 Dwarn improvement over the other policies (small architecture) . . . 44
3.20 Dwarn improvement over the other policies (deep architecture) 45

4.1 Main tasks of a resource allocation policy 51
4.2 Schema of I-Fetch and RAlloc policies 54
4.3 Comparison of the different resource allocations for ILP and MEM workloads 56
4.4 Hmean results for all resource allocations for all MIX workloads . . . 57
4.5 IPC throughput 59
4.6 Hmean improvement of smartRA over various I-fetch policies 59

4.7 Increment in the number of fetched instructions when the flush policy is used . . . 60
4.8 (a) QoS space for three fetch policies; (b) important QoS points and areas . . . 62
4.9 QoS space, IPC values, and overall throughput for all workload types 64

5.1 Average IPC of SPEC benchmarks as we vary the amount given to them when the data L1 cache is perfect. . . . 71
5.2 Possible implementation of our dynamic allocation policy . . . 75
5.3 Throughput/Hmean results of dcra compared to static resource allocation . . . 79
5.4 Throughput/Hmean improvement of dcra over icount, flush++, and dg. . . . 80
5.5 Dcra Hmean improvement over other mechanisms as we change the register pool size. . . . 81
5.6 Dcra Hmean improvement over the other mechanisms as we vary the IQs. 82
5.7 Hmean improvement of dcra over other mechanisms as we change the memory latency. . . . 82

6.1 OS/SMT collaboration . . . 91
6.2 IPC of `gzip` for different contexts and different fetch policies . . . 92
6.3 IPC of the *gzip* benchmark for different handicap values . . . 93
6.4 QoS space for three fetch policies and important QoS points and areas 95
6.5 Local IPC of the `gap` benchmark . . . 96
6.6 Sample and Tune phases . . . 97
6.7 Misses suffered by the PPT due to the UPTs interference . . . 98
6.8 Realized IPC values for the PPT. The x-axis shows the target percentage of full speed of the PPT and size of the workloads. The four different bars represent the four different types of workload discussed in Section 6.3.4 . . . 105
6.9 QoS space where the y-axis shows the target percentage of full speed of the PPT, and the x-axis the throughput obtained by our mechanism with respect to *flush++*. The legend shows different workload types as well as target percentages required for the PPT. . . . 107
6.10 QoS space for all workload types . . . 110
6.11 Shared resources required to achieve a given relative IPC . . . 111
6.12 Total throughput with respect to *flush++* . . . 113
6.13 UPTs throughput : (a)XXI workloads; (b)XXM workloads . . . 115
6.14 Relation between the amount of resources given to a task and its IPC. 121
6.15 Cache interference introduced by the NCT as we vary the amount of resources given to the CT. . . . 124
6.16 Variability in CT's IPC in different workloads as we vary the resources under control. . . . 126
6.17 Different performance/resources functions . . . 128

6.18 Success Rate and throughput as we change the p/r relation 130
6.19 Success Rate and throughput of our approach and previous approaches 131
6.20 Hardware required to implement our mechanism 132

List of Tables

2.1 FastForward used for each Spec CPU 2000 Benchmark. 13
2.2 Cache behavior of each benchmark in a 512Kb L2 cache 14
2.3 MediaBench Benchmarks used in this thesis. 14

3.1 Response Action \ Detection Moment space 20
3.2 Baseline configuration 22
3.3 Workload classification based on cache behavior of threads. 23
3.4 Baseline configuration 34
3.5 Workload classification based on cache behavior of threads. 34
3.6 Cache behavior of each benchmark 36
3.7 Relative IPC of each thread in the 4-MIX workload 42

4.1 Resource allocation used 52
4.2 Resource actions and input information of fetch policies 53
4.3 Baseline configuration 55
4.4 Workload classification based on cache behavior of threads. 55
4.5 Issue queue and physical register division that lead to the best Hmean results. 58

5.1 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. C=1/T. 74
5.2 Baseline configuration 77
5.3 Workload classification based on cache behavior of threads. 77
5.4 Distribution of threads in phases for 2-thread workloads 79

6.1 FSM to track phases 102
6.2 Baseline configuration 103
6.3 Workloads for 1-thread prioritization 104
6.4 Subset of workloads for 2-thread prioritization (condensed) 108
6.5 Workloads for 2-thread prioritization 109
6.6 Baseline configuration 119
6.7 Comparing approaches shown in this chapter 134

Bibliography

[1] http://cares.icsl.ucla.edu/mediabench/.

[2] http://www.cpuid.org/k8/index.php.

[3] http://www.intel.com/design/network/products/npfamily/.

[4] http://www.research.ibm.com/journal/rd/472/allen.html.

[5] IA-32 intel architecture software developer's manual. volume 3: System programming guide.

[6] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A processor architecture for multiprocessing. Technical Report MIT/LCS/TM-450, 1991.

[7] Don Alpert. Will microprocessor become simpler? *Microprocessor Report*, Nov 2003.

[8] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. pages 1–6, 1990.

[9] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. Virtual simple architecture (visa): Exceeding the complexity limit in safe real-time systems. *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 350–361, June 2003.

[10] Max Baron. Two threads for tricore 2. *Microprocessor Report*, Sep 2003.

[11] J. Burns and J-L. Gaudiot. Exploring the SMT fetch bottleneck. *Proceedings of the 3rd Workshop on Multithreaded Execution, Architecture, and Compilation*, 1999.

[12] J. Burns and J-L. Gaudiot. Quantifying the SMT layout overhead- does SMT pull its weight? *Proceedings of the 6th International Conference on High Performance Computer Architecture*, pages 109–120, January 2000.

[13] 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*, pages 70–85, October 2003.

[14] F.J. Cazorla, P.M.W. Knijnenburg, E. Fernandez, R. Sakellariou, A. Ramirez, and M. Valero. Implicit vs. explicit resource allocation in SMT processors. *In EUROMICRO Symposium on Digital System Design. Invited Paper*, pages 44–51, 2004.

[15] F.J. Cazorla, P.M.W. Knijnenburg, E. Fernandez, R. Sakellariou, A. Ramirez, and M. Valero. Predictable performance in SMT processors. *ACM International Conference on Computing Frontiers*, pages 433–443, 2004.

[16] F.J. Cazorla, P.M.W. Knijnenburg, E. Fernandez, R. Sakellariou, A. Ramirez, and M. Valero. Qos for high-performance SMT processors in embedded systems. *IEEE micro. Special Issue on Embedded Systems*, 24(4):24–31, July/August 2004.

[17] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. *Proceedings of Design Automation Conference*, June 2000.

[18] G. Dorai and D. Yeung. Transparent threads: Resource sharing in SMT processors for high single-thread performance. *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, pages 30–41, Sep 2002.

[19] G. Dorai, D. Yeung, and S. Choi. Optimizing SMT processors for high single-thread performance. *Journal of ILP*, 5, April 2003.

[20] A. El-Moursy and D.H. Albonesi. Front-end policies for improved issue efficiency in SMT processors. *Proceedings of the 9th International Conference on High Performance Computer Architecture*, pages 65–76, February 2003.

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

[22] A. Falcon, O. J. Santana, A. Ramirez, and M. Valero. Selecting where to simulate spec2000 using stream analysis. *In Proceedings of the XV Jornadas de Paralelismo*, September 2004.

[23] Ayose Falcon, Alex Ramirez, and Mateo Valero. A low complexity, high-performance fetch unit for simultaneous multithreading processors. *Proceedings of the 10th International Conference on High Performance Computer Architecture*, February 2004.

[24] Semiconductor Industry Association (SIA). International Technology Roadmap for Semiconductors 2001. http://public.itrs.net/files/2001itrs/. 2001.

[25] J. Fritts, W. Wolf, and B. Liu. Understanding multimedia application characteristics for designing programmable media processors. *SPIE Photonics West - Media Processors*, pages 2–13, January 1999.

[26] 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 13rd Symposium on Computer Architecture and High Performance Computing*, Sep 2001.

[27] 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, February 1996.

[28] T. R. Halfhill. Philips powers up for video. *Microprocessor Report*, November 2003.

[29] R. Halstead and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. pages 443–451, May 1988.

[30] S. Heo, K. Barr, M. Hampton, and K. Asanovic. Dynamic fine-grain leakage reduction using leakage-biased bitlines. *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 131–147, May 2002.

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

[32] 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.

[33] C. J. Hughes, P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan. Variability in the execution of multimedia applications and implications for architecture. *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 254–265, 2001.

[34] R. Jain, C.J. Hughes, and S.V. Adve. Soft real-time scheduling on simultaneous multithreaded processors. *Proceedings of the 23rd International Symposium on Real-Time Systems Symposium*, pages 134–145, Dec 2002.

[35] N. P. Jouppi. Cache write policies and performance. Technical Report 91/12, digital. Western Research Laboratory, 1991.

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

[37] P.M.K. Knijnenburg, A. Ramirez, J. Larriba, and M. Valero. Branch classification for SMT fetch gating. *Proceedings of the 6th Workshop on Multithreaded Execution, Architecture, and Compilation*, pages 49–56, 2002.

[38] K. Krewell. Fujitsu makes SPARC see double. *Microprocessor Report*, November 2003.

[39] J. T. Kuehn and B. J. Smith. The horizon supercomputing system: Architecture and software. pages 28–34, November 1988.

[40] A. Lee, M. Potkonjak1, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 330–335, 1997.

[41] Markus Levy. Multithreaded technologies disclosed at MPF. *Microprocessor Report*, Nov 2003.

[42] L. Li, I. Kadayif, Y-F. Tsai, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, and A. Sivasubramaniam. Leakage energy management in cache hierarchies. *In proceedings of the IEEE 11th International Conference in Parallel Architectures and Compilation Techniques*, pages 131–140, September 2002.

[43] 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.

[44] J. L. Lo. Exploiting thread-level parallelism on simultaneous multithreaded processors. *Ph. D. Thesis. University of Washington*, 1998.

[45] K. Luo, M. Franklin, S.S. Mukherjee, and A. Seznec. Boosting SMT performance by speculation control. *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 23–27, April 2001.

[46] 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.

[47] 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.

[48] D. Mulvihill and M. Allen. Evaluating branch predictors on an SMT processor. Technical Report CS 752, University of Wisconsin–Madison, 2002.

[49] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for SMT processors. Technical Report 2000-04-02, Department of Computer Science and Engineering. University of Washington, 2000.

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

[51] M. Ramsay, C. Feucht, and M. H. Lipasti. Exploring efficient SMT branch predictor design. *WCED*, June 2003.

[52] S. Sair and M. Charney. Memory behavior of the spec'2000 benchmark suite. Technical Report RC-21852, IBM Thomas J. Watson Research Center, October 2000.

[53] Y. Sazeides and T. Juan. How to compare the performance of two SMT microarchitectures. *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, November 2001.

[54] M. J. Serrano, R. Wood, and M. Nemirovsky. A study on multistreamed superscalar processors. Technical Report 93-05, University of California Santa Barbara, 1993.

[55] A. Settle, J. L. Kihm, A. Janiszewski, and D. A. Connors. Architectural support for enhanced SMT job scheduling. *Proceedings of the 13rd International Conference on Parallel Architectures and Compilation Techniques*, pages 63–73, October 2004.

[56] 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*, pages 3–14, September 2001.

[57] R. Shin, S.-W. Lee, and J. L. Gaudiot. Dynamic scheduling issues in SMT architectures. *Proceedings of the International Parallel and Distributed Processing Symposium*, Apr 2003.

[58] B. Smith. Architecture and applications of the HEP multiprocessor computer system. *In Proceedings of the Fourth Symposium on Real Time Signal Processing*, pages 241–249, 1981.

[59] A. Snavely, D.M. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreaded processor. *ACM SIGMETRICS*, pages 234–244, June 2002.

[60] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE micro*, 22(4):72–82, July 2002.

[61] S. Storino, A. Aipperspach, J. Borkenhagen R. Eickemeyer, S. Kunkel, S. Levenstein, and G. Uhlmann. A commercial multithreaded risc processor. pages 234–235, February 1998.

[62] S. Swanson, L. McDowell, M. Swift, S. Eggers, and H. Levy. An evaluation of speculative instruction execution on simultaneous multithreaded processors. *ACM Transactions on Computer Systems*, 21(3):314–1340, 2003.

[63] 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*, pages 318–327, December 2001.

[64] 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 23rd Annual International Symposium on Computer Architecture*, pages 191–202, April 1996.

[65] D.M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *Proceedings of the 22rd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[66] T. Ungerer, B. Robic, and J. Silc. Multithreaded processors. *The Computer Journal*, 45(3):320–341, November 2002.

[67] T. Ungerer, B. Robic, and J. Silc. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35(1):29–63, March 2003.

[68] David W. Wall. Limits of instruction-level parallelism. *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 1991.

[69] Synopsys white paper. Managing power in ultra deep submicron asic/ic design. *In Proceedings of the 2003 international symposium on Low power electronics and design*, May 2002.

[70] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual ISCA*, pages 84–97, June 2003.

[71] 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.

[72] 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*, pages 42–53, May 1999.