

Static Partitioning vs Dynamic Sharing of Resources in Simultaneous MultiThreading Microarchitectures

Chen Liu

*Department of Electrical Engineering and
Computer Science
University of California, Irvine
cliu3@uci.edu*

Jean-Luc Gaudiot

*Department of Electrical Engineering and
Computer Science
University of California, Irvine
gaudiot@uci.edu*

Abstract

Simultaneous MultiThreading (SMT) achieves better system resource utilization and higher performance because it exploits Thread-Level Parallelism (TLP) in addition to Instruction-Level Parallelism (ILP). Theoretically, system resources in every pipeline stage of an SMT microarchitecture can be dynamically shared. However, in commercial application, all the major queues are statically partitioned. From an implementation point of view, static partitioning of resources is easier to implement with lower hardware overhead and less power consumption. In this paper, we strive to quantitatively determine the tradeoff between static partitioning and dynamic sharing. We find that static partitioning of either the instruction fetch queue (IFQ) or the reorder buffer (ROB) is not sufficient if implemented alone (3% and 9% performance decrease respectively in the worst case comparing with dynamic sharing), while statically partitioning both the IFQ and ROB could achieve an average performance gain of 9% at least, 148% when running with floating point benchmarks, comparing with dynamic sharing. We varied the number of functional units in our efforts to isolate the reason for this performance improvement. We found that static partitioning both queues outperforms all the other partitioning mechanisms as long as under same system configuration. This demonstrates that the performance gain has been achieved by moving from dynamic sharing to static partitioning of the system resources.

I. INTRODUCTION

Simultaneous MultiThreading (SMT) has been a hot research area for more than one decade. From the initial implementation in the CDC 6600, the HEP, the TERA, the HORIZON, and the APRIL architectures, in which there

exists some concept of multi-threading or Simultaneous MultiThreading, to the real commercial implementation of SMT in the latest Pentium 4 and XEON processor family with HyperThreading (HT) technology, all demonstrates the power of SMT (another commercial design of SMT, the COMPAQ EV8, was abandoned even before reaching the manufacturing stage). Because of the limitations of Instruction-Level Parallelism (ILP), the performance gain that traditional superscalar processors could achieve is diminishing even with an increase in the number of execution units. On the other hand, through issuing and executing instructions from multiple threads at every clock cycle, SMT could achieve maximum system resource utilization and higher performance.

However, when it comes to the problem of how to allocate the system resources to the multiple threads, there are different opinions. It is sometimes assumed that dynamic sharing on system resources at every pipeline stage in the SMT microarchitectures, which means threads can compete for the resources and there is no quota on the resources that one single thread could utilize, could be as low as 0%, or could be 100%. In other cases, all the major queues are statically partitioned, so each thread has its own portion of the resources and there is no overlapping in between.

From the implementation point view, static partitioning of resources is easier to implement with less hardware overhead, which matches exactly INTEL's implementation goal of hyperthreading, "smallest hardware overhead and high enough performance gain"; on the other hand, dynamic sharing could maximize the utilization of the system resources and corresponding performance, even though with higher hardware costs and power consumption.

The goal of this paper is to quantify the impact of static partitioning vs dynamic sharing on the overall performance of the system. We study the effect of different partitioning mechanisms (static partitioning vs dynamic sharing) on the

different system resources (instruction fetch queue and reorder buffer, for example), and their impact on the overall system performance.

Prior to our proposed work, we review related work of different partitioning methods on the system resources in Section II. Section III describes our experiment methodology. Our simulated work is discussed in more detail in Section IV. Conclusions are presented in Section V.

II. RELATED WORK

Marr *et al.* presented the commercial implementation of a 2-thread SMT architecture in INTEL’s XEON processor family. In their implementation, almost all the queues are statically divided into two, one for each thread. However, the scheduler queues are shared by both threads so that the schedulers could dispatch instructions to the execution engine regardless of which thread they come from, to insure the timely execution in order to maintain a high throughput. However, there is still a cap on the number of instructions one thread could have in scheduler queues. When we come to the reason why choosing the static-partitioning implementation, it was not unveiled in the context.

An investigation of the impact of different system resource partitioning mechanisms on SMT processors was performed by Raasch *et al.*. In this paper, various system resources, like instruction queue, reorder buffer, issue bandwidth, commit bandwidth are studied under different partitioning mechanisms. The authors draw the conclusion that the true power of SMT lies in the ability to issue instructions from different threads in one clock cycle. Hence, the issue bandwidth has to be shared all the time. While different partitioning mechanisms on other system resources like storage queues will result in very little impact on the system performances. However, their work is mainly focused on the back-end of the pipeline, *e.g.*, execution and retirement part, did not affect any of the front-end of the pipeline, *e.g.*, the fetch part. We extended their work by studying the different partitioning techniques on the front-end instruction fetch queue and the back-end reorder buffer, as well as their impact on the overall performance.

III. OUR APPROACH

When we study the pipeline from front to end, we see there are many system resources which could be under different partitioning mechanisms, for example, the instruction fetch queue, the instruction decode queue, the instruction issue queue (sometimes called instruction queue), the reorder buffer, the load/store queue, etc. In our proposed work we selected two resources from above, the

front-end instruction fetch queue (IFQ) and the back-end reorder buffer (ROB), and applied different partitioning mechanisms on them separately. Then, we compared the performance of each configuration to find out the impact of different partitioning mechanisms on the overall system performance, which is measured in terms of Instruction per Cycle (IPC). This comparison would lead us to get the optimum configuration. Here we listed all 4 combinations of architectures to simulate for a 2-thread Simultaneous MultiThreading architecture:

- SMT: Both the front-end instruction fetch queue and the back-end reorder buffer are in dynamic sharing mode, same as other system resources
- SIFQ: Only the front-end instruction fetch queue is divided into two, one for each thread, and other system resources are in dynamic sharing mode
- SROB: Only the back-end reorder buffer is divided into two, one for each thread, and other system resources are in dynamic sharing mode
- STOUS: Both the front-end instruction fetch queue and the back-end reorder buffer are divided into two, one for each thread, and other system resources are in dynamic sharing mode.

In each configuration, we perform extensive simulations to get the average system performance.

IV. SIMULATION

To properly evaluate the effects of the proposed partitioning mechanism, we designed an execution-driven simulator, based on an SMT simulator developed by Kang *et al.*, which is itself derived from SimpleScalar, through modifying the *sim-outorder* simulator to implement an SMT processor model. Following the structure of *sim-outorder*, the architectural model contains seven pipeline stages: fetch, decode, dispatch, issue, execute, complete, and commit. Several resources, such as program counter (PC), integer and floating-point register files, and branch predictor, are replicated to allow multiple thread contexts. The simulator uses the 64-bit PISA instruction set.

A. Experiment Setup

The major simulator parameters are listed in Table 1. The fetch policy employed is Instruction Count (IC). The simulator is configured to issue as many instructions as the total number of functional units at each clock cycle according to the priority set by the IC policy.

Table 1. Simulation parameter

Parameter	Value
Instruction fetch rate	8
Instruction decode rate	8

Instruction retire rate	8
L1instruction Cache	64Kbytes (256:64:4:LRU)
L1 data Cache	64Kbytes (512:32:4:LRU)
L2 Cache	1Mbytes (2048:128:4:LRU)
Memory Access Bus Width	32 bytes
Instruction TLB	512Kbytes (32:4096:4:LRU)
Data TLB	1Mbytes (64:4096:4:LRU)
Instruction Issue Queue Size	64
LQ/SQ size	64/64
INT units	8
FP units	4

The simulator has been modified to accommodate the changes of the corresponding sharing policy for IFQ and ROB. In table 2 we listed the corresponding instruction fetch queue size and the reorder buffer size for each configuration.

Table 2. Simulation setup

Configuration Name	Instruction Fetch Queue Size	Reorder Buffer Size
SMT	256	256
SIFQ	128 128	256
SROB	256	128 128
STOUS	128 128	128 128

The benchmarks used are all from SPEC CPU2000 benchmark suite [6]. The ten benchmarks used (7 integers and 3 floating-point benchmarks) are listed in Table 3.

Table 3. SPEC2000 CPU Benchmark using in the simulation

Benchmark	Type	Language	Category
164.gzip	INT	C	Compression
175.vpr	INT	C	FPGA Circuit Placement and Routing
176.gcc	INT	C	C Programming Language Compiler
179.art	FP	C	Image Recognition / Neural Networks
181.mcf	INT	C	Combinatorial Optimization
183.quake	FP	C	Seismic Wave Propagation Simulation
188.amp	FP	C	Computational

197.parser	INT	C	Chemistry Word Processing
256.bzip2	INT	C	Compression
300.twolf	INT	C	Place and Route Simulator

Since there are 10 sets of benchmarks, 4 sets of simulator configuration for the 2-thread input, we run each benchmark with all the benchmarks (including itself). Hence altogether we run 4*10*10 iterations of simulation to get all the results. Each iteration of simulation is composed of 1000 million instructions, after fast forwarding through the first 300 million instructions from each thread to skip the initialization part of the benchmark. Then the results (IPC) are averaged to get the average performance for each benchmark under each partitioning configuration.

B. Simulation Result

In Fig. 1, we present the average performance in term of IPC for each benchmark under different partitioning architectures. Obviously STOUS architecture outperforms other partitioning architectures.

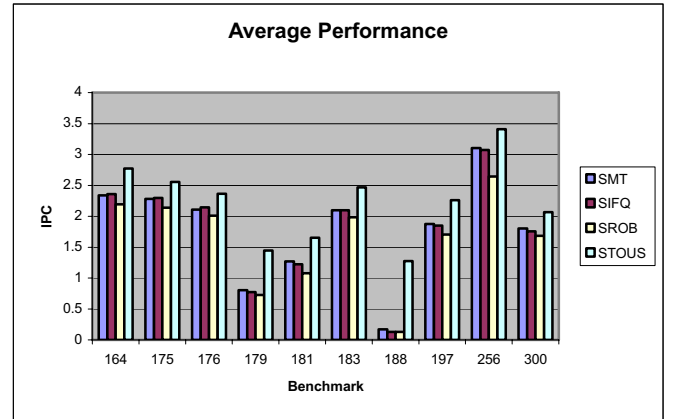


Fig. 1 Average performance gain for different partitioning architectures

We exploit the following formulas to compute the performance gain:

$$Gain1 = \frac{IPC_{SIFQ} - IPC_{SMT}}{IPC_{SMT}} \quad (1)$$

$$Gain2 = \frac{IPC_{SROB} - IPC_{SMT}}{IPC_{SMT}} \quad (2)$$

$$Gain3 = \frac{IPC_{STOUS} - IPC_{SMT}}{IPC_{SMT}} \quad (3)$$

When we examine the results more carefully using the above formulas, we observe that when Benchmark 179 and 188 runs together with other benchmarks, they could achieve such a huge performance gain (up to 7 or 8 fold), which may exaggerate the performance gain achieved from other benchmarks. Therefore, in Table 4 we list the average performance gain in three different situations:

- Overall average performance gain, which is computed using the results of running all 10 benchmarks
- Average performance gain excluding Benchmark 188, which is computing using only the results from running the rest 9 benchmarks
- Average performance gain excluding Benchmark 179 and 188, which is computed using only the results from running the rest 8 benchmarks

The reason is because we want to examine the performance gain excluding the interference from those two benchmarks, to see how other benchmarks react to the different system partitioning architectures. From the table, we can see STOUS architecture keeps giving positive performance gain, while other architectures could result in performance loss.

Table 4. Performance gain

	Gain1 (%)	Gain2 (%)	Gain3 (%)
Overall Average Performance Gain	19.69	-6.94	148.25
Average Performance Gain excluding Benchmark 188	-0.10	-7.66	23.67
Average Performance Gain excluding Benchmark 188 and 179	-3.35	-8.66	8.99

C. Impact of functional units

In order to isolate the reason why Benchmark 188 and 179 could achieve such huge performance gain, we redo the

simulation by varying the number of INT and FP functional units as listed in Table 5. We also increased the size of instruction issue queue, load queue/store queue from 64 entries to 128 entries.

Table 5 Functional units configuration

Configuration	I	II	III
INT units	4	8	8
FP units	4	4	8

In Fig. 2-4, we can see the average IPC for each partitioning mechanism with different functional unit configuration.

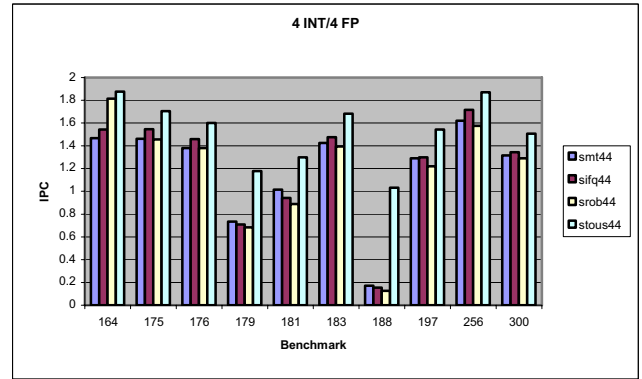


Fig. 2 Average IPC for 4 INT / 4 FP functional units configuration

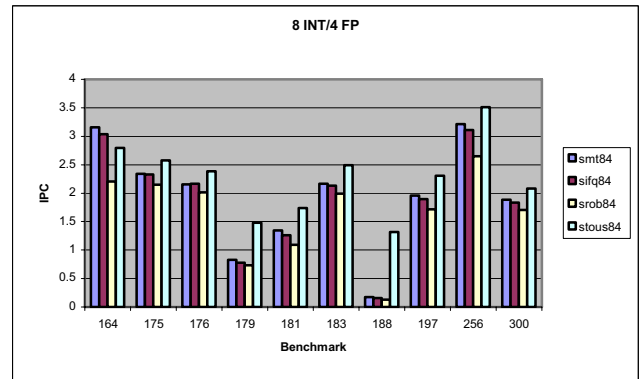


Fig. 3 Average IPC for 8 INT / 4 FP functional units configuration

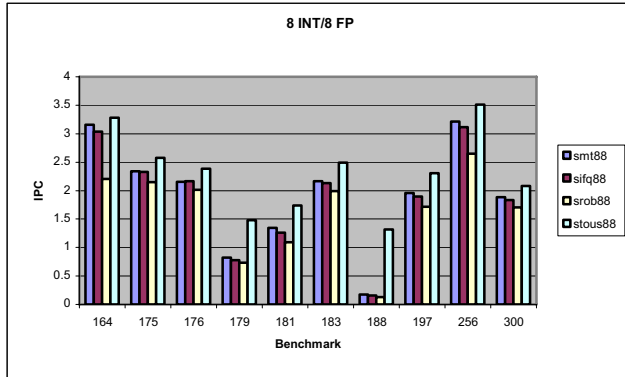


Fig.4 Average IPC for 8 INT / 8 FP functional units configuration

From the graph, we can see that with different variations in the number of functional units, the STOUS architecture keeps outperforming SMT, SIFQ and SROB architectures in term of IPC. This proves that the performance gain is achieved from the difference between static partitioning and dynamic sharing of the system resources, not because of the number of functional units in favor of any of the architectures.

V. CONCLUSION

From the above tables and graphs, several conclusions can be drawn:

1. Static partitioning either IFQ or ROB solely can only lead to negative performance gain.
2. Static partitioning both the IFQ and ROB together could achieve marginal performance gain (even in worst scenario when running integer benchmark solely, STOUS architecture could still achieve 10% performance gain over SMT architecture).

We feel that the reason for this is that static partitioning both the IFQ and ROB is like forcing the input and output of the pipeline to evenly execute the two input threads, avoiding the situation where one of the threads grabs more resources it could use and clog the pipeline, while the other thread could not get enough resources and under-executed. Only static partitioning either one of them could not achieve such result because it only controls one end of the pipeline while have no control over the other end.

The huge performance gain from running Benchmark 188 and 179 results from better system resource utilization with one integer and one floating-point input. Because now the instructions from different thread are running on separate functional units, minimize the competition for those resources, maximize the throughput, which shows the original power of SMT.

Through the static partitioning the instruction fetch queue and reorder buffer, we could achieve better performance (in terms of IPC) than dynamic sharing. And at the same time, static partitioning would require less hardware overhead, and also achieve less power consumption.

Since the inter-functioning of IFQ and ROB could bring us this opportunity to achieve better performance with less complicated static partitioning mechanism, then the next step work is to try the different partitioning mechanisms on other system resources, to study the inter-relationship among them, and in the end to find an optimum way to sharing system resources to achieve the best performance with the least hardware overhead and power consumption.

	Name:	Liu, Chen (刘晨)
	Contact Information:	Department of Electrical Engineering and Computer Science University of California Irvine, CA 92697
	Phone:	949-824-9748
	Email:	Clui3@uci.edu
Education background:	<i>Ph.D candidate</i> (09/2002 - present) Department of Electrical Engineering and Computer Science University of California Irvine, CA 92697	
	<i>Master of Science</i> Department of Electrical Engineering University of California, Riverside (UCR) 09/2000 – 08/2002	
	<i>Bachelor of Engineering</i> Department of Electrical Engineering and Information Science University of Science and Technology of China (USTC) 09/1995 – 07/2000	
Research Interest:	Computer architecture, general processor design, multi-threading architecture	