Static Partitioning vs Dynamic Sharing of Resources in Simultaneous MultiThreading Microarchitectures

Chen Liu and Jean-Luc Gaudiot

Department of Electrical Engineering and Computer Science,
University of California, Irvine, CA 92697, USA
{cliu3, gaudiot}@uci.edu
http://pascal.eng.uci.edu

Abstract. Simultaneous MultiThreading (SMT) achieves better system resource utilization and higher performance because it exploits Thread-Level Parallelism (TLP) in addition to "conventional" Instruction-Level Parallelism (ILP). Theoretically, system resources in every pipeline stage of an SMT microarchitecture can be dynamically shared. However, in commercial applications, all the major queues are statically partitioned. From an implementation point of view, static partitioning of resources is easier to implement and has a lower hardware overhead and power consumption. In this paper, we strive to quantitatively determine the trade-off 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 the ROB could achieve an average performance gain of 9% at least, and even reach 148% when running with floating-point benchmarks, when compared 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 outperformed all the other partitioning mechanisms under the same system configuration. This demonstrates that the performance gain has been achieved by moving from dynamic sharing to static partitioning of the system resources.

1 Introduction

Simultaneous MultiThreading (SMT) has been a hot research area for more than one decade [14,15,16,17,18,19,20,21]. From the embryonic implementation in the CDC 6600 [22], the HEP [9], the TERA [8], the HORIZON [12], and the APRIL [13] architectures, in which there exists some concept of multi-threading or Simultaneous MultiThreading, to the actual commercial implementation of SMT in the latest Pentium 4 [10] and XEON [5] processor families with HyperThreading (HT) technology, all demonstrates the power of SMT (another commercial design of SMT, the COMPAQ EV8 [11], 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 can 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. Sometimes the dynamic sharing method is applied on system resources at every pipeline stage in the SMT microarchitectures [16,17,18] (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 [4,5], so that each thread has its own portion of the resources and there is no overlap.

From the implementation point view, static partitioning of resources is easier to implement with lower hardware overhead and less power consumption, which matches exactly with INTEL's implementation goal of hyperthreading – smallest hardware overhead and high enough performance gain [5]. On the other hand, dynamic sharing is normally assumed to be able to maximize the utilization of the system resources and corresponding performance, even though it would come at a higher hardware cost and more power consumption.

The goal of this paper is thus 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 overall system performance.

Prior to our proposed work, we review related work of different partitioning methods on the system resources in Section 2. Section 3 describes our experiment approach. Our simulated work is discussed in more detail in Section 4. Conclusions are presented in Section 5.

2 Related Work

Marr *et al.* [5] presented a 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 can dispatch instructions to the execution engine regardless of which thread they come from, so as to insure timely execution and maintain a high throughput. However, there is still a cap on the number of instructions one thread could have in scheduler queues.

An investigation of the impact of different system resource partitioning mechanisms on SMT processors was performed by Raasch *et al.* in [1]. In this paper, various system resources, like instruction queue, reorder buffer, issue bandwidth, and commit bandwidth are studied under different partitioning mechanisms. The authors conclude that the true power of SMT lies in its 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 the impact on the overall performance caused by the interaction between them.

3 Our Approach

There are many system resources in a pipeline 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 four combinations of architectures to simulate a 2-thread Simultaneous Multi-Threading architecture:

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

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

4 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.* [7], which is itself derived from SimpleScalar [3], 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.

4.1 Experiment Setup

The major simulator parameters are listed in Table 1. The fetch policy employed is Instruction Count (I-Count). 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 I-Count policy.

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.

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

Since there are 10 sets of benchmark, 4 sets of simulator configuration for the 2-thread input, we run each benchmark with all the benchmarks (including

Table 1. Simulation parameters

Parameter	Value
Instruction Fetch Rate	8
Instruction Decode Rate	8
Instruction Retire Rate	8
L1 Instruction 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

Table 2. Simulation setup

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

Table 3. SPEC2000 CPU Benchmark used 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.equake	FP	C	Seismic Wave Propagation Simulation
188.ammp	FP	C	Computational Chemistry
197.parser	INT	C	Word Processing
256.bzip2	INT	C	Compression
300.twolf	INT	C	Place and Route Simulator

itself). Hence altogether we run $4 \times 10 \times 10$ iterations of simulation to get all the results. Each iteration of simulation is composed of 1 billion 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.

4.2 Simulation Results

In Fig. 1, we present the average performance in term of IPC for each benchmark under different partitioning architectures. Obviously, the STOUS architecture outperforms other partitioning approaches. We derived the following formulas to compute the performance gain:

$$Gain1 = \frac{IPC_{SIFQ} - IPC_{SMT}}{IPC_{SMT}} \tag{1}$$

$$Gain2 = \frac{IPC_{SROB} - IPC_{SMT}}{IPC_{SMT}} \tag{2}$$

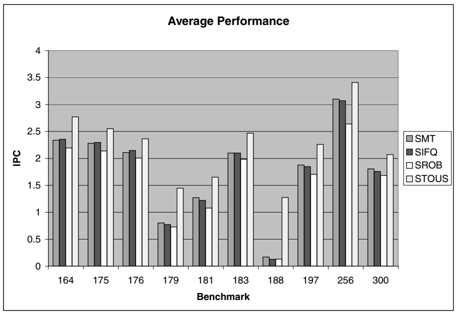

Fig. 1. Average performance gain for different partitioning architectures

Table 4. Performance Gain Comparison

	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 179 and 188)	-3.35	-8.66	8.99

$$Gain3 = \frac{IPC_{STOUS} - IPC_{SMT}}{IPC_{SMT}} \tag{3}$$

When we examine the results more carefully under the light of the above formulas, we observe that when Benchmark 179 and 188 run together with other benchmarks, they could achieve such a huge performance gain (up to 7 or 8 fold), that they 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 remaining nine benchmarks.
- Average performance gain excluding Benchmark 179 and 188, which is computed using only the results from running the remaining eight benchmarks.

The reason why we want to compare the performance under these three different situations is because we want to examine the performance gain excluding the interference from those two benchmarks (179 and 188), to see how other benchmarks react to the different system partitioning architectures. From the table, we can see that the STOUS architecture keeps yielding positive performance gain, while other architectures could result in a loss of performance.

4.3 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

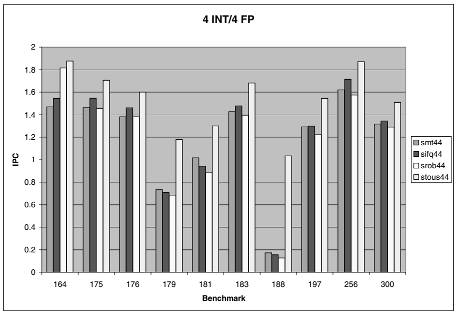

Fig. 2. Average IPC for 4 INT / 4 FP functional units configuration

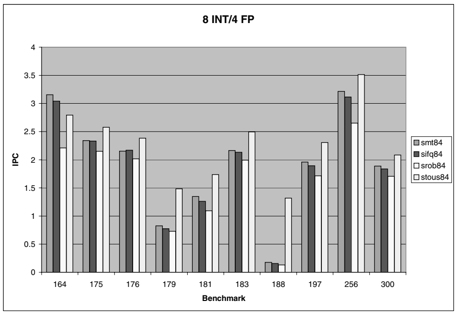

Fig. 3. Average IPC for 8 INT / 4 FP functional units configuration

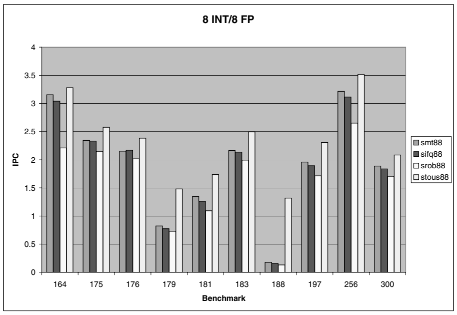

Fig. 4. Average IPC for 8 INT / 8 FP functional units configuration

In Fig. 2, 3, 4, we can see the average IPC for each partitioning mechanism with different functional unit configuration.

From the graph, we can see that with different variations in the number of functional units, the STOUS architecture keeps outperforming the SMT, SIFQ and SROB architectures in term of IPC. This demonstrates 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.

5 Conclusions

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

1. Statically partitioning either the IFQ or the ROB solely can only lead to a negative performance gain.
2. Statically partitioning both the IFQ and ROB together could achieve marginal performance gain (even in the worst scenario when running integer benchmarks solely, the STOUS architecture could still achieve 9% performance gain over the SMT architecture).

We feel 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. Hence we can avoid the situation where one of the threads grabs more resources it could use and clogs the pipeline, while the other thread could not get enough resources and under-executed. Statically partitioning either one of them could not achieve such results because it only controls one end of the pipeline while there is 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 in stead of competing with each other for the same type of functional units, the instructions from different threads are running on different types of functional units. Hence the competition for those resources is minimized, the throughput is maximized, which shows the original power of SMT.

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

Since static partitioning of both IFQ and ROB could bring us this opportunity to achieve better performance with a less complicated mechanism, then the next step is to try 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 for the SMT microarchitecture.

Acknowledgment. We would like to thank Dongsoo Kang for his support in our use of his SMT simulator.

References

1. Raasch, Steven E., Reinhardt, Steven K.: The Impact of Resource Partitioning on SMT Processors. Proceedings of the 12th Intenrational Conference on Parallel Architectures and Compilation Techniques (PACT 2003). New Orleans, Louisiana, USA. Sep. 27 - Oct. 01. (2003) 15–26
2. Sazeides, Y., Juan, T.: How to Compare the Performace of Two SMT Microarchitectures. Proceedings of 2001 IEEE International Symposium on Performance Analysis of System and Software (ISPASS-2001). Tucson, Arizona, USA, November 4-6. (2001)
3. Burger, D., Austin, T.: The SimpleScalar Tool Set, Version 2.0. University of Wisconsin-Madison Computer Science Department Technical Report No.1342. June (1997)
4. Koufaty, D., Marr, Deborah T.: Hyperthreading Technology in the Netburst Microarchitecture. IEEE Micro, March-April. (2003)
5. Marr, Deborah T., Binns, F., Hill, David L., Hinton, G., Koufaty, David A., Miller, J.Alan, Upton, M.: Hyper-Threading Technology Architecture and Microarchitecture. Intel Technology Journal Q1. (2002)
6. SPEC CPU 2000 Benchmark Suite: http://www.specbench.org/osg/cpu2000/ (2000)
7. Kang, D., Gaudiot, J-L.: Speculation control for simultaneous Multithreading. Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004). Santa Fe, New Mexico, April 26-30.(2004)
8. Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A., Smith, B.: The TERA Computer System. ACM SIGARCH Computer Architecture News, Vol. 18, No. 3.(1990) 1–6
9. Smith, B.J.: Architecture and Applications of the HEP Multiprocessor Computer System. SPIE Real Time Signal Processing IV. (1981) 241-248
10. Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D, Kyker, A., Roussel, P.: The Microarchitecture of the Pentium 4 Processor. Intel Technology Journal Q1. (2001)
11. Preston, Ronald P., Badeau, Roy W., Bailey, Daniel W., Bell, Shane L., *et al.*: Design of an 8-wide Superscalar RISC Microprocessor with Simultaneous Multithreading. Proceedings of 2002 IEEE International Solid-State Circuits Conference (ISSCC 2002). Vol. 1. (2002)
12. Thistle, Mark R., Smith, Burton J.: A Processor Architecture for HORIZON. Proceedings of the 1988 ACM/IEEE Conference on Supercomputing. Orlando, Florida, USA, Nov. 12-17. (1988) 35–41
13. Agarwal, A., Lim, B-H., Kranz, D., Kubiatowicz, J.:APRIL: A Processor Architecture for Multiprocessing. Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA 1990). (1990) 104–114
14. Nemirovsky, Mario D., Brewer, F., Wood, Roger C.: DISC: Dynamic Instruction Stream Computer. Proceedings of the 24th annual international symposium on Microarchitecture (Micro-24). Albuquerque, New Mexico, Puerto Rico. (1991) 163–171
15. Yamamoto, W., Nemirovsky, Mario D.: Increasing superscalar performance through multistreaming. Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques. Limassol, Cyprus. (1995) 49–58
16. Tullsen, Dean M., Eggers, Susan J., Levy, Henry M.: Simultaneous Multithreading: Maximizing On-chip Parallelism. Procedding of the 22nd Annual International Symposium on Computer Architecture (ISCA 1995). (1995) 392–403

17. Tullsen, Dean M., Eggers, Susan J., Emer, Joel S., Levy, Henry M., Lo, Jack L., Stamm, Rebecca L.: Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA 1996). (1996) 191–202
18. Eggers, Susan J., Emer, Joel S., Levy, Henry M., Lo, Jack L., Stamm, Rebecca L., Tullsen, Dean M.: Simultaneous Multithreading: A Platform for Next-Generation Processors. IEEE Micro, Vol. 17. No. 5. (1997) 12–19
19. Shin, C-H., Lee, S-W., Gaudiot, J-L.: Dynamic Scheduling Issues in SMT Architectures. Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03). Nice, France, April 22-26. (2003) 77-84
20. Burns, J., Gaudiot, J-L.: SMT Layout Overhead and Scalability. IEEE Transactions on Parallel and Distributed Systems, Vol. 13, No. 2. (2002) 142–155
21. Lee, S-W., Gaudiot, J-L.: Clustered Microarchitecture Simultaneous Multithreading. Proceedings of the Euro-Par 2003 International Conference on Parallel and Distributed Computing, Klagenfurt, Austria, August 26-29. (2003)
22. Thornton, J. E.: Design of a computer: the CDC 6600. Scott, Foresman Co., Glenview, Ill. (1970)