The Impact of Resource Sharing Control on the Design of Multicore Processors

Chen Liu[1] and Jean-Luc Gaudiot[2]

[1] Department of Electrical and Computer Engineering,
Florida International University, 10555 West Flagler Street,
Miami FL 33174, USA
cliu@fiu.edu

[2] Department of Electrical Engineering and Computer Science,
University of California, Irvine CA 92697, USA
gaudiot@uci.edu

Abstract. One major obstacle faced by designers when entering the multicore era is how to harness the massive computing power which these cores provide. Since Instructional-Level Parallelism (ILP) is inherently limited, one single thread is not capable of efficiently utilizing the resource of a single core. Hence, Simultaneous MultiThreading (SMT) microarchitecture can be introduced in an effort to achieve improved system resource utilization and a correspondingly higher instruction throughput through the exploitation of Thread-Level Parallelism (TLP) as well as ILP. However, when multiple threads execute concurrently in a single core, they automatically compete for system resources. Our research shows that, without control over the number of entries each thread can occupy in system resources like instruction fetch queue and/or reorder buffer, a scenario called "mutual-hindrance" execution takes place. Conversely, introducing active resource sharing control mechanisms causes the opposite situation ("mutual-benefit" execution), with a possible significant performance improvement and lower cache miss frequency. This demonstrates that active resource sharing control is essential for future multicore multithreading microprocessor design.

1 Introduction

With multicore microprocessors being introduced into commercial desktop and laptop computers, parallel computing has become mainstream in the computing community. While 2-core and 4-core microprocessors still dominate the market, projections and early prototypes already indicate that we could soon be looking at hundred, even thousand cores on a single chip in the near future. Since Instructional-Level Parallelism (ILP) for a single thread is inherently limited, we believe future multi/many-core microprocessor should have the capability of multithreading. Hence, in this article, we focus on the design of Multicore Multithreading MicroProcessor (MMMP). At first glance, it might appear that scheduling threads to execute on an MMMP is as straightforward as simply following existing techniques similar to scheduling multiple tasks onto multiple single-core microprocessors, which does not presenting a substantially difficulty. The truth is, however, how to achieve efficient utilization of these many processing elements and harness the tremendous computing power of future multi/many-core processors presents a great challenge to system architect.

In MMMP, the resource sharing control scheme has a significant impact on overall system performance and power efficiency. The scheme must decide how the system resources on each core are divided among multiple threads. For example, how many entries one thread can occupy in Instruction Fetch Queue (IFQ), Instruction Issue Queue (IIQ), ReOrder Buffer (ROB), Renaming Register, separately. How to divide the issue and commit bandwidth among multiple threads is also a part of the scheme. If there is no control on the resources that can be assigned to one thread in one core, this would cause the uneven distribution of resources among threads and uneven execution of the threads. This also translates into the overall time to execute all threads being extended and more power being consumed in this prolonged process. On the other hand, an active resource sharing control scheme will alleviate these symptoms or even avoid them from happening. Our goal is to achieve sustained computational throughput at an acceptable level so that the overall time to execute multiple threads is reduced and less power is consumed. This would require all the threads progress at a similar rate, if not the same, which can be achieved through resource sharing control. In addition, through resource sharing control, we can even satisfy the required Quality-of-Service (QoS) criteria for real-time applications [1]. Hence, resource sharing control of an MMMP is a fundamentally different and much more difficult problem than that of Single-core Single-thread MicroProcessor (SSMP) design.

The purpose of this paper is to validate the necessity of an active resource management scheme for future MMMP design. We demonstrate the effectiveness of such a scheme in a Single-core Multithreading MicroProcessor (SMMP) design, and argue that this design can be easily imported into future MMMP design. In section 2 we introduce some background on microprocessor design and related research. Our experiments allow us to frame the issue related to resource sharing control in section 3. In section 4 we discuss future design trends and conclude.

2 Background

With the emergence of multicore processors, a new prediction after Moore's law [2] has been born, stating that instead of doubling the number of transistors, it is the number of cores on a single chip that will double every 18 months from now on [3]. While putting a large number of cores on the same chip, however, we still face the same "Memory Wall" problem [4] as in the single-core era. This is one of the motivations of our work. As such, attempting to exploit Thread-Level Parallelism (TLP) is an effort to overcome the limitation due to low ILP within a single program. There are two ways to explore TLP: multicore architectures and multithreading architectures.

2.1 Multicore and Multithreading

As a result of a number of limiting factors including VLSI layout, architectural design, and more importantly, the dilemma between power and frequency, the general-purpose processor architecture paradigm is facing a change. The otherwise too-complex too-high-frequency single-core processor design has to be abandoned. With an abundance of transistor real estate, we are turning to improving the performance by having multiple low-complexity cores on the same die. High-performance processor design is rapidly moving towards many-core architectures that integrate tens (or more) of processing cores on a single chip, represented by Intel® Teraflops [5], IBM Cyclops-64 [6] and Nvidia® Tesla$^{\text{TM}}$ architectures.

There are two different architectural technologies for multicore processor design. One is to integrate tightly-coupled identical processing elements on a single chip, this is called homogeneous multicore, represented by the Intel® Core$^{\text{TM}}$ 2 Duo and the AMD Turion$^{\text{TM}}$ 64 X2. The other is to integrate a number of different processing elements (some of them maybe application specific) on a single chip, called heterogeneous multicore, represented by the IBM Cell Broadband Engine$^{\text{TM}}$. The many-core-on-a-chip architecture naturally exploits the Thread-Level and Process-Level Parallelism, which will enable tomorrow's emerging applications and is expected to be supported by multicore-aware operating systems (OS) and environments.

By allowing one processor to execute two or more threads concurrently, Simultaneous MultiThreading architecture is able to exploit both Instructional-Level Parallelism and Thread-Level Parallelism, accordingly achieving improved instruction throughput [7,8]. Hence, Simultaneous MultiThreading is one of the best architectures for utilizing the vast computing power that such a microprocessor would have, achieving optimal system resource utilization and higher performance. For this reason and also because of the limited ILP from a single thread, SMT is a good architectural technique which can maximize system performance.

2.2 Resource Allocation and Management

Researchers have certainly realized the importance of active resource allocation for an SMT processor design in order to achieve the optimum resource distribution among threads, which leads to desired performance improvement. Indeed, Raash et al. [9] studied various system resource partitioning mechanisms on SMT processors. They concluded that the true power of SMT lies in its ability to issue and execute instructions from different threads at every clock cycle, hence, the issue bandwidth has to be shared all the time. Incidentally, it should be noted that we certainly incorporated this philosophy into our design. Fetch policy can achieve implicit resource distribution among threads with very limited effect. For example, ICOUNT policy [8] prioritizes the threads based on the number of instructions in the front-end stages from each thread to decide from which thread to fetch instructions. DCRA [10] was proposed in an attempt to dynamically allocate the resources among threads by dividing the execution of each thread into different phases, using instruction and cache miss counts as indicators. The study shows that DCRA achieves around 18% performance gain over ICOUNT in terms of harmonic mean. Hill-Climbing [11] dynamically allocates the resources based on the current performance of each thread and feedback into the resource-allocation engine. It used its learning-based algorithm to selectively sample different resource distributions first to find out the local optimum and then adopt that distribution. It achieves a slightly higher performance (2.4%) than DCRA but is certainly the most expensive one in terms of execution overhead. ARPA [12] is proposed as an adaptive resource partitioning algorithm that dynamically assigns resources (IFQ, IQ and ROB) to the threads according to thread behavior changes. ARPA analyzes the resource usage efficiency of each thread in a time period and assigns more resources to threads which can use them more efficiently. It outperforms Hill-Climbing scheme by 5.7%.

Some other recent work has explored various design approaches, which all requires a closer interaction between OS and the multicore microarchitecture, including resource management. Based on their work in fault-handling and thread-scheduling in SMT processor design, Bower et al. [13] argued that under the heterogeneous multicore processor scenario, we should schedule threads by pairing the thread with the core which best matches its execution demand. Elevating this decision-making process up to the OS level would be even more beneficial. Knauerhase et al. [14] basically agrees with this approach in the MMMP design. They suggested that the OS should be able to dynamically migrate the thread onto a different core, so as to achieve the ideal scheduling of pairing the cache-heavy and cache-light threads on the same core. This would improve overall system performance, improve application performance and decrease system power consumption, similar to the "mutual-benefit" and "mutual-hindrance" scenarios we address in this paper. Nesbit et al. [15] introduced Virtual Private Machine (VPM) into the resource management of multicore processors. Their goal is to elevate the resource management to the software level, not solely at the hardware level. After the application specifying the QoS objective, the system software translates these objectives into hardware resource assignment, e.g., the number of cores and the portion of shared L2 cache to be dedicated to this application solely, through a software/hardware interface based on VPM abstraction. In this way, the QoS objective is satisfied.

3 Impact of Resource Sharing Control

Here we use a set of simple experiments to demonstrate that resource sharing control can provide a substantial advantage in terms of overall performance improvement and weighted speedup over no-control schemes. However, before that, we will briefly introduce our evaluation methodology.

3.1 Evaluation Methodology

Our simulator is based on the SMT simulator developed by Kang et al. [16], which itself is derived from SimpleScalar [17]. Our simulator is meant to

Table 1. Baseline parameters

Parameter	Value
IF, ID, IR Width	8-way
Queue Size	64 IIQ, 64 LQ, 64 SQ
Functional Units	6 INT, 2 FP, 4 Load/Store
	2 INT Mul/Div, 2 FP Mul/Div
Instruction Fetch Queue Size	256 entries
ReOrder Buffer Size	256 entries
BTB	512 entries, 4-way associative
Branch Predictor	Hybrid: 1K gshare + 1K bimodal
L1 D-Cache	64KB, 4-way
L1 I-Cache	64KB, 4-way
Combined L2 Cache	1MB, 4-way associative
L2 Cache Hit Latency	6 cycles
Main Memory Hit Latency	100 cycles

Table 2. Nine SPEC CPU2000 benchmarks used in this study

Benchmark	Type	Character.	Benchmark	Type	Character.	Benchmark	Type	Character.
mcf	INT	MEM	gcc	INT	ILP	equake	FP	MEM
parser	INT	MEM	gzip	INT	ILP	art	FP	MEM
twolf	INT	MEM	bzip2	INT	ILP			
vpr	INT	MEM						

simulate an SMT architectural model which supports out-of-order and speculative execution. The major baseline SMT simulation parameters are listed in Table 1.

Table 2 lists the nine benchmarks we used in our simulations. All benchmarks are from the SPEC CPU2000 suite [18] with the reduced input set from MinneSPEC [19]. Following the same methodology as shown in [10,11,12], we divide the benchmarks into two different categories: memory-bound (we refer to them as MEM for short) and computation-bound (ILP for short). Based on these characteristics of the benchmarks, we created appropriately mixed 2-thread multiprogramming workloads shown in Table 3. For this study we only focus on the 2-thread SMT architecture.

Measuring the performance of multiprogramming workloads on a multithreading processor can be tricky. We cannot only look at the overall instruction throughput; we also need to consider fairness execution for each thread. In this paper we use two metrics to evaluate both throughput and fairness as in [11,12,20]. The first metric is *avg_IPC*, which is defined as:

$$avg\_IPC = \frac{\sum_{i=1}^{N} IPC_i}{N} \tag{1}$$

Table 3. Workload combinations based on benchmark behavior

Name	Combination	Name	Combination	Name	Combination
MEM1	vpr, art	MEM6	art, mcf	MEM11	mcf, parser
MEM2	vpr, mcf	MEM7	art, equake	MEM12	mcf, twolf
MEM3	vpr, equake	MEM8	art, parser	MEM13	equake, parser
MEM4	vpr, parser	MEM9	art, twolf	MEM14	eqauke, twolf
MEM5	vpr, twolf	MEM10	mcf, equake	MEM15	parser, twolf
MIX1	gzip, vpr	MIX7	gcc, vpr	MIX13	bzip2, vpr
MIX2	gzip, art	MIX8	gcc, art	MIX14	bzip2, art
MIX3	gzip, mcf	MIX9	gcc, mcf	MIX15	bzip2, mcf
MIX4	gzip, equake	MIX10	gcc, equake	MIX16	bzip2, equake
MIX5	gzip, parser	MIX11	gcc, parser	MIX17	bzip2, parser
MIX6	gzip, twolf	MIX12	gcc, twolf	MIX18	bzip2, twolf
ILP1	gzip, gcc	ILP2	gzip, bzip2	ILP3	gcc, bzip2

Where N is the number of threads, for our case $N = 2$. Note that *avg_IPC* only measures the overall throughput, which may be biased under certain circumstances like executing more instructions from the thread with fewer long-latency instructions while executing fewer instructions from the thread having more long-latency instructions. Hence, we also need another metric to take fairness into consideration:

$$avg\_Baseline\_Weighted\_IPC = \frac{\sum_{i=1}^{N} \frac{IPC_{new,i}}{IPC_{baseline,i}}}{N} \tag{2}$$

avg_Baseline_Weighted_IPC weighs the IPC of each thread in the new scheme with respect to its IPC in the baseline scheme and then calculates the average. This takes the relative performance improvement of all threads into consideration, which is a fairer measurement.

3.2 Performance Evaluation

In our previous work [21], we proposed four different resource sharing control schemes:

1. **D-Share**, where all the system resources are dynamically shared by both threads.
2. **IFQ-Fen**, where one thread can only hold up to half of the Instruction Fetch Queue (IFQ) entries and all other system resource are dynamically shared.
3. **ROB-Fen**, where one thread can only hold up to half of the ReOrder Buffer (ROB) entries and all other system resource are dynamically shared.
4. **Dual-Fen**, where both IFQ and ROB are monitored and one thread can only hold up to half of the entries of each queue. If any thread is beyond its quota, the fetch from that thread is temporarily suspended until the situation is resolved.

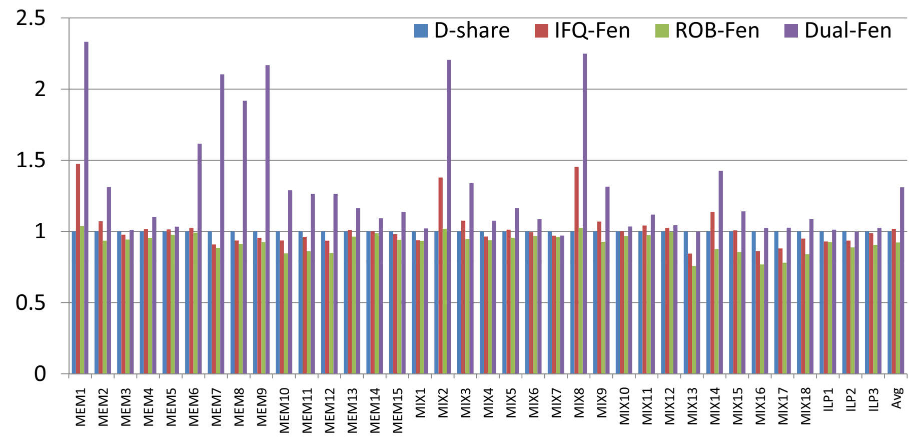

Fig. 1. avg_Baseline_Weighted_IPC of different schemes for 36 workloads

This paper is a continuation of our previous work [21]. We strive to evaluate the impact of resource sharing control schemes on the performance of SMMP from a different perspective, by introducing a new set of evaluation metrics and criteria.

Figure 1 compares *avg_Baseline_Weighted_IPC* of all the schemes proposed across the 36 workloads listed in Table 3, using D-Share as the baseline configuration. We can see that the Dual-Fen scheme achieves a better speedup over the D-Share scheme for almost all workloads in MEM and MIX groups; the improvement in the ILP group is not significant compared with that in the MEM and MIX groups. Overall, IFQ-Fen, ROB-Fen and Dual-Fen achieve 1.9%, −7.7% and 31.0% improvements over D-Share for 2-thread workloads.

All the resource sharing control schemes run on top of the ICOUNT policy, which is capable to indicate the current performance of the thread to some extent. It, however, does not take speculative execution into consideration, nor does it handle long latency (mainly cache miss) instructions well. If no preventive measure is taken, the instructions from the thread with a cache miss will occupy system resources in the pipeline. This directly translates into a reduction in the amount of system resources that other thread(s) can utilize. This is what we call "mutual hindrance" execution. Because the memory-bound threads in MEM and MIX workloads more easily clog the pipeline than do computation-bound threads in ILP workloads, we can see from Figure 1 that the improvement in MEM and MIX workloads is much greater than that in ILP workloads. Resource sharing control can prevent one thread from grabbing too many system resources. For this case, our Dual-Fen scheme controls both the input to the pipeline and its output in order to achieve what we call "mutual-benefit" execution. In this way, we have control over the resource distribution, reduce the impact of cache misses on the pipeline and enhance the overall system performance.

Figure 2 shows the *avg_IPC* improvement and the *avg_Baseline_Weighted_IPC* improvement on average for the MEM, MIX and ILP workloads separately. With *avg_IPC* metric, IFQ-Fen, ROB-Fen and Dual-Fen achieve −0.2%, −7.9% and

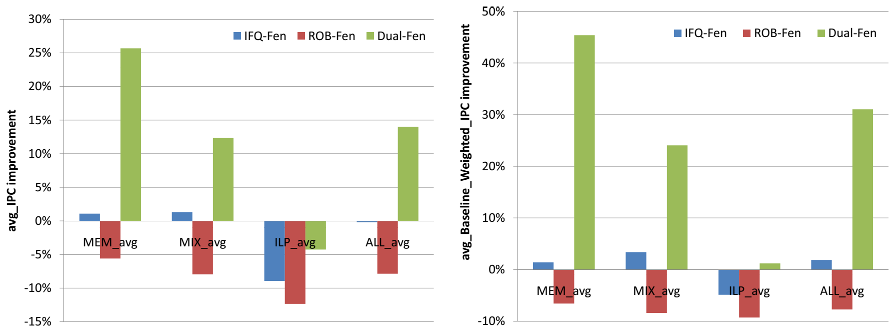

Fig. 2. avg_IPC and avg_Baseline_Weighted_IPC improvement of different schemes over D-Share across 36 workloads

14.0% improvements over D-Share, respectively. The *avg_Baseline_ Weighted_IPC* improvement for Dual-Fen and IFQ-Fen over D-Share is much better than the *avg_IPC* improvement for the MEM and MIX groups. For example, Dual-Fen achieves a 24% improvement in *avg_Baseline_Wei-ghted_IPC* over D-Share for MIX_avg, but using the *avg_IPC* metric, the improvement is only 12%. This is caused by the different characteristics of the metrics.

Dual-Fen performs much better than D-Share, IFQ-Fen and ROB-Fen schemes in MEM and MIX workloads but shows no significant advantages for ILP workloads. The reason is that computation-bound threads normally requires fewer resources to exploit ILP and not prone to cache miss instructions, while memory-bound threads requires more resources to exploit ILP, with more cache misses and clogging the pipeline. Dual-Fen scheme shows the capability of effectively controlling the resource utilization by potentially clogging thread to improve the overall throughput, while ROB-Fen scheme is not able to achieve the same level of control. Since a clogging thread which uses clogged resources for a long period of time usually has a low IPC reading for the dynamic sharing case, resource sharing control scheme such as Dual-Fen greatly reduces the extend of this kind of clogging from happening, which resulted in a greater weighted IPC improvements.

Some benchmarks, like *art*, achieve a better performance improvement than others. There are two reasons for that. One is that *art* is of floating-point type. When executing side-by-side with an integer-type thread, it is not competing for the same type of functional units, and hence more TLP can be explored. Another reason, is that *art* itself when running alone is prone to cache misses. With our resource sharing control scheme Dual-Fen, the L1 D-Cache miss rate for *art* has been greatly reduced compared with D-Share case when running with another thread. This proves that our resource sharing control scheme effectively reduces the extent of clogging of the pipeline caused by long-latency operations. Also because the reduction in cache-miss rate, the overall time that the instruction spends in the pipeline (slip-time) has also been reduced on average by 34%, as shown in [21].

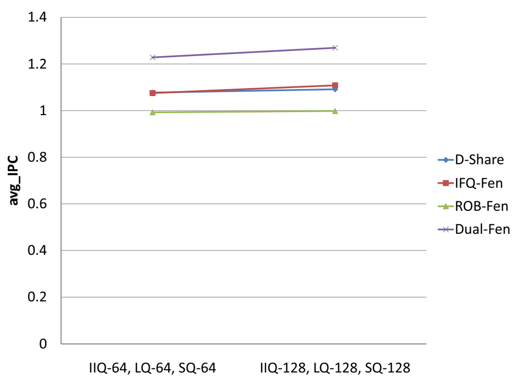

Fig. 3. avg_IPC of different schemes as the number of queue entries changes across 36 workloads

3.3 Sensitivity Analysis

In this section, we study the impact of the queue size and functional units on resource sharing control. Figure 3 shows the average IPC of Dual-Fen versus D-Share, IFQ-Fen and ROB-Fen schemes across the 36 workloads as the Instruction Issue Queue (IIQ), Load Queue (LQ) and Store Queue (LQ) are all increased from 64 entries to 128 entries with the IFQ and ROB remaining at 256 entries. We can see all schemes only achieve minor performance improvement. Actually increasing the IIQ, LQ and SQ size almost has no impact on ROB-Fen and D-Share. IFQ-Fen and Dual-Fen performance improves by 3%. With more entries in the queue, more ILP can be exploited when searching for ready instructions to execute within limited extent. While ROB-Fen still under-performs D-Share, which means controlling the ROB alone cannot achieve the desired resource distribution.

Figure 4 compares the *avg_Baseline_Weighted_IPC* improvement of different schemes over D-Share as the functional units of the pipeline increase from (4-INT, 4-FP) to (8-INT, 8-FP) across all 36 workloads as the IIQ, LQ and SQ sizes are fixed at 128 entries. We can see that as the number of functional unit increases, the performance of Dual-Fen scheme over D-share is the biggest among all the schemes, achieving 43%, 37% and 38%, respectively, while IFQ-Fen only achieves 3%, 6% and 10% improvements. ROB-Fen scheme keeps under-performing. When the number of functional units increases from (4-INT, 4-FP) to (8-INT, 4-FP), we see the IFQ-Fen performance is improved by 3% while the Dual-Fen drops by 4%. However, when the number of functional units increases from (8-INT, 4-FP) to (8-INT, 8-FP), we see the IFQ-Fen performance is improved by another 4% while the Dual-Fen only improved by 1%.

We believe the reason for that is when we increase the number of integer functional units, the benchmark with more ILP can benefit from this change because now its instructions can be issued for execution at a faster speed. Since IFQ-Fen only controls the input to the pipeline, and the increase of the function

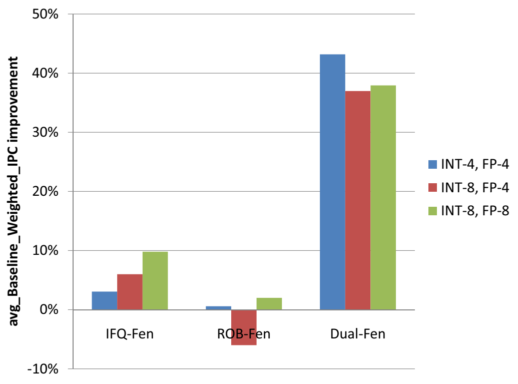

Fig. 4. avg_Baseline_Weighted_IPC improvement of different schemes over D-Share as the number of functional unit changes across 36 workloads

units mainly affects the execution and retirement stages of the pipeline, which is why we see a performance jump for IFQ-Fen. As for Dual-Fen, because it also controls the ROB, which affects the instruction retirement speed. This is why we do not observe a performance improvement of Dual-Fen as the number of functional units increases, as opposed to the IFQ-Fen case. This can be deemed as a trade-off for avoiding the clogging of the pipeline from cache misses. Overall, Dual-Fen scheme still achieves a huge performance improvement over all other schemes, just it does not benefit from the extra functional units added.

4 Conclusion and Future Design Trends

From a number of experiments we have shown that resource sharing control is essential for SMMP design, which naturally extends to MMMP design as well. Under a multithreading execution scenario, one of the evaluation metrics on which we focus is the overall system throughput, which translates into overall execution time in the completion of multiple-thread workload. Through active resource sharing control, first of all, we can avoid the scenario that one thread grabs too many resources and then clog the pipeline when a long-latency operation happens. Secondly, this will also contribute towards closing the execution speed gap between the fast-moving thread and slow-moving thread, moving towards an even-execution of both threads. Under this "mutual-benefit" execution scenario, we could achieve the maximum TLP, with the resource sharing control schemes controlling the ILP of each thread.

As we have seen, resource sharing control scheme does not work well when computation-bound thread running together with a similar type thread. So if we can get help from OS scheduler, when scheduling the job, always trying to pairing a computation-bound thread with a memory-bound thread onto one core, this would definitely improve the overall system performance in MMMP. What's more, each thread's behavior changes as it passes through phases of execution.

Built on top of the resource sharing control scheme for each core, we may need support from OS to hot-swab the thread from one-core to another if the thread is in a different phase of execution and we want to keep matching one computation-bound thread with one memory-bound thread, as long as the benefits justify the swabbing overhead. This would greatly benefit the performance of future MMMPs, even satisfy certain QoS specifics.

We mean for this paper to demonstrate the urgent need for resource sharing control for multicore multithreading microprocessors. Controlling resource sharing for this new type of microprocessor is a new problem, which requires knowledge of the dynamic status of cores and the dynamic characteristics of threads. This would require that the operating system designer, the compiler designer, and the hardware architect collaborate accordingly. We will need to construct an interface to better communicate across the different layers in order to achieve this goal.

References

1. Cazorla, F., Ramirez, A., Valero, M., Knijnenburg, P., Sakellariou, R., Fernandez, E.: QoS for high-performance SMT processors in embedded systems. IEEE Micro 24(4), 24–31 (2004)
2. Moore, G.E.: Cramming more components onto integrated circuits. Electronics 38(8), 114–117 (1965)
3. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, University of California at Berkeley (2006)
4. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 4th edn. Morgan Kaufmann Publishers Inc., San Francisco (2006)
5. Vangal, S., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Singh, A., Jacob, T., Jain, S., Erraguntla, V., Roberts, C., Hoskote, Y., Borkar, N., Borkar, S.: An 80-tile sub-100-w teraflops processor in 65-nm CMOS. IEEE Journal of Solid-State Circuits 43(1), 29–41 (2008)
6. Zhang, Y.P., Jeong, T., Chen, F., Wu, H., Nitzsche, R., Gao, G.: A study of the on-chip interconnection network for the IBM Cyclops64 multi-core architecture. In: IPDPS 20: Proceedings of the 20th International Parallel and Distributed Processing Symposium, p. 44. IEEE Computer Society, Los Alamitos (2006)
7. Tullsen, D., Eggers, S., Levy, H.: Simultaneous multithreading: Maximizing on-chip parallelism. In: ISCA 22: Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 392–403. IEEE Computer Society Press, Los Alamitos (1995)
8. Tullsen, D., Eggers, S., Emer, J., Levy, H., Lo, J.L., Stamm, R.: Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In: ISCA 23: Proceedings of the 23rd Annual International Symposium on Computer Architecture, p. 191. IEEE Computer Society, Los Alamitos (1996)
9. Raasch, S., Reinhardt, S.: The impact of resource partitioning on SMT processors. In: PACT 2003: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, pp. 15–25. IEEE Computer Society, Los Alamitos (2003)

10. Cazorla, F.J., Ramirez, A., Valero, M., Fernandez, E.: Dynamically controlled resource allocation in SMT processors. In: MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, pp. 171–182. IEEE Computer Society, Los Alamitos (2004)
11. Choi, S., Yeung, D.: Learning-based SMT processor resource distribution via hill-climbing. In: ISCA 2006: Proceedings of the 33rd Annual International Symposium on Computer Architecture, pp. 239–251. IEEE Computer Society, Los Alamitos (2006)
12. Wang, H., Koren, I., Krishna, C.M.: An adaptive resource partitioning algorithm for SMT processors. In: PACT 2008: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 230–239. ACM Press, New York (2008)
13. Bower, F.A., Sorin, D.J., Cox, L.P.: The impact of dynamically heterogeneous multicore processors on thread scheduling. IEEE Micro 28(3), 17–25 (2008)
14. Knauerhase, R., Brett, P., Hohlt, B., Li, T., Hahn, S.: Using OS observations to improve performance in multicore systems. IEEE Micro 28(3), 54–66 (2008)
15. Nesbit, K.J., Moreto, M., Cazorla, F.J., Ramirez, A., Valero, M., Smith, J.E.: Multicore resource management. IEEE Micro 28(3), 6–16 (2008)
16. Kang, D., Liu, C., Gaudiot, J.L.: The impact of speculative execution on SMT processors. International Journal of Parallel Programming 36(4), 361–385 (2008)
17. Burger, D., Austin, T.M.: The simplescalar tool set, version 2.0. ACM SIGARCH Computer Architecture News 25(3), 13–25 (1997)
18. Henning, J.L.: SPEC CPU 2000: Measuring CPU performance in the new millennium. Computer 33(7), 28–35 (2000)
19. KleinOsowski, A.J., Lilja, D.J.: MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. IEEE Computer Architecture Letters 1(1), 7 (2002)
20. Luo, K., Gummaraju, J., Franklin, M.: Balancing thoughput and fairness in SMT processors. In: ISPASS 2001: Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software, pp. 164–171 (2001)
21. Liu, C., Gaudiot, J.L.: Resource sharing control in simultaneous multithreading microarchitectures. In: ACSAC 2008: Proceedings of the 13th Asia-Pacific Computer Systems Architecture Conference, pp. 1–8 (2008)