The Impact of Dynamically Heterogeneous Multicore Processors on Thread Scheduling

Journal:	*IEEE Micro*
Manuscript ID:	MicroSI-2008-01-0004
Manuscript Type:	Special Issue May/June 08 Interaction of Computer Architecture and OS (submissions due 5 Jan 2008)
Date Submitted by the Author:	04-Jan-2008
Complete List of Authors:	Bower, Fred; Duke University, Computer Science Sorin, Daniel J.; Duke University, ECE Cox, Landon; Duke University, Computer Science
Keywords:	C.0.b Hardware/software interfaces < C.0 General < C Computer Systems Organization, C.0.a Emerging technologies < C.0 General < C Computer Systems Organization, C.1.2 Multiple Data Stream Architectures (Multiprocessors) < C.1 Processor Architectures < C Computer Systems Organization, C.1.4.e Multi-core/single-chip multiprocessors < C.1.4 Parallel Architectures < C.1 Processor Architectures < C Computer Systems Organization

The Impact of Dynamically Heterogeneous Multicore Processors on Thread Scheduling

Fred A. Bower[1,2], Daniel J. Sorin[3], and Landon P. Cox[2]

bowerf@us.ibm.com, sorin@ee.duke.edu, lpcox@cs.duke.edu

[1]IBM Systems and Technology Group, System x Development

[2]Duke University, Department of Computer Science

[3]Duke University, Department of Electrical and Computer Engineering

Contact author: Daniel J. Sorin
PO Box 90291
Durham, NC 27708
phone: 919-660-5439
fax: 919-660-5293
sorin@ee.duke.edu

Abstract

The computer industry has turned to multicore processors as a power-efficient way to use the vast number of transistors on a chip. Most current multicore processors are homogeneous (i.e., all the cores are identical), and scheduling them is similar, but not identical, to what operating systems have been doing for years. However, microarchitects are proposing heterogeneous core implementations, including systems in which heterogeneity is introduced at runtime. These processors will require schedulers that can adapt to heterogeneity.

In this position paper, we discuss the trends that are leading to dynamic heterogeneity, and we show that schedulers must consider dynamic heterogeneity or suffer significant power-efficiency and performance losses. We present the challenges posed by dynamic heterogeneity, and we argue for research into hardware and software support for efficiently scheduling such systems.

1 Introduction

Moore's Law provides computer architects with more transistors than they can effectively use to extract instruction level parallelism in a single core. Thus, all current and future high-performance processor chips are *multicore processors* (also known as *chip multiprocessors* or *CMPs*). These multicore processors include the Cell Broadband Engine [11], Intel's CoreDuo and Quad-Core Xeon, AMD's Dual-Core Opteron, Sun Microsystems' Niagara [13], and IBM's Power5 [12]. These processors have between two and eight cores in a single chip package, with the expectation of greater numbers of cores in future generations.

At first glance, scheduling a multicore processor may not appear to present a substantially new problem for operating systems. There is a long history of OS scheduling for multithreaded microprocessors and traditional multi-chip multiprocessors. There has even been recent research [9, 8] into one aspect of scheduling that is unique to multicores, which is that processors often share L2 caches. Except for this issue of cache sharing, it might at first appear that scheduling of multicore processors would be a straightforward extension of existing scheduling techniques, *except future multicore processors are unlikely to be composed of homogeneous cores* [15, 1]. Due to core specialization and runtime fault handling and power management, it is likely that multicore processors will feature heterogeneous cores. Furthermore, this heterogeneity is likely to be both static (as an intentional design feature that does not change) and dynamic (as a response to run-time events such as physical faults or power management). In this position paper, we focus on dynamically heterogeneous multicore processors (DHMPs), because they will present a greater challenge to future operating systems.

In a DHMP, the OS scheduler has a significant impact on power efficiency and performance. The scheduler must decide which threads (or portions of threads) should run on which cores. A good schedule will match each thread with a core that can provide it with sufficient performance at an acceptable power cost. A poor schedule will match each thread with a core that either cannot run it at an acceptable performance (e.g., due to faults in that core) or that needlessly wastes power running it. A poor schedule can lead to shorter battery life for a laptop, slower response time for a gaming console, greater power costs for small business computing, or less computational throughput for a server farm. Scheduling a DHMP is a fundamentally different and more difficult problem than scheduling homogeneous systems.

In the rest of this paper we further motivate and define the research challenges presented by DHMPs. Section 2 reviews multicore architecture to frame the issues related to scheduling these systems. In Section 3, we present the challenges we see for the efficient scheduling of DHMPs. In Section 4, we discuss the limited research that has been done in this area. We conclude in Section 5 with a call to action for scheduling research, outlining fruitful future areas of research.

2 Multicore Trends and Impact

With the increasing transistor budgets afforded by Moore's Law, architects have sought ways to use all of them in a power-efficient manner. Until recently, architects have dedicated their transistor budget to extracting instruction level parallelism (ILP) out of single-threaded code. However, dedicating current transistor budgets strictly to ILP is not power-efficient, and thus architects have sought to use transistors to also exploit thread level parallelism. An initial approach was simultaneous multithreading (SMT) [18], such as in the Pentium4, in which multiple threads share a single core's resources. Unfortunately, a single SMT processor is also limited in how many transistors it can use power-efficiently. Thus, the industry has begun placing multiple independent cores on each chip. Each of these cores may itself be multithreaded, providing a multiplicative number of schedulable contexts.

Homogeneous multicore processors are composed of identical cores that provide a consistent computing capability for each schedulable context. Homogeneity simplifies the job of the scheduler and allows us to use existing scheduling algorithms for multiprocessor systems. These algorithms may factor cache warmth and cache sharing into scheduling decisions, but they generally do not discriminate amongst cores, as they are all viewed as equally capable of performing computations.

2.1 Sources of Heterogeneity

The primary problem with homogeneous multicore processors is that naive replication of state-of-the-art single-core designs in a single package (or chip package) stress the power and cooling limits for the chip. There is a fixed amount of power that a chip can consume before we cannot cool it (with air cooling). Given this power budget, a chip cannot contain dozens of Pentium4-like processors, even if each one is itself power-efficient. Nevertheless, for high-priority tasks, we still want the single-threaded performance provided by current high-performance cores. Thus, architects believe that (statically) heterogeneous multicore designs, such as the Cell Processor [11], will be prevalent in coming generations [15, 1]. As illustrated in Figure 1, a multicore design might consist of a small number of high-power and high-performance cores, coupled with a

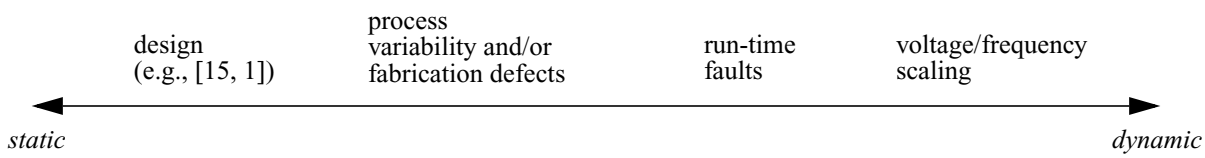

Figure 2. Sources of Heterogeneity. We characterize sources of heterogeneity by how frequently they affect the processing capability of the core. Fully static designs have a fixed set of core capabilities that can be advertised via specification. At the fully-dynamic end of the spectrum, core capabilities may change every scheduling quantum, requiring scheduler adaptability in order to effectively exploit the heterogeneity.

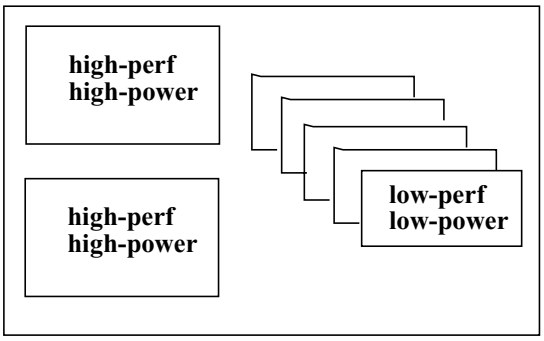

Figure 1. Statically Heterogeneous Multicore

larger number of simpler, low-power cores. These low-power cores will be tailored to providing computing power for the background tasks for which a user may tolerate greater latency.

In addition to static heterogeneity, we also expect dynamic heterogeneity because of at least three technological issues. First, fabrication process variability is increasing [3] and it is highly likely that the cores will have different performance characteristics. Thus, it will be preferable to clock the cores at different frequencies, rather than derate the entire chip to the lowest-common operating frequency, particularly as core counts continue to increase. This form of heterogeneity is dynamic, in that it is not known at design-time, but it is fixed once the chip has been fabricated and tested.[1]

Second, with the increasing numbers of transistors per chip and the trends towards increasing physical fault rates per transistor [17], it is therefore likely that cores will have to be able to reconfigure themselves to tolerate faults. Our prior research [4, 5] has explored how to detect and diagnose permanent faults in a single core. In response to these faults, part of the core may be deconfigured, resulting in the core moving to a lower-performing state, but still providing useful work to the system.[2] This sort of deconfiguration is fairly rare and it can either occur at manufacturing time (to address manufacturing faults) or during the part's lifetime (to address lifetime reliability issues). If we apply this approach or a similar technique to a multicore, even one that is designed to be homogeneous, it will lead to a DHMP.

Third, each core is likely to incorporate its own dynamic voltage and frequency scaling (DVFS). DVFS techniques are in use today, but are constrained to chip-wide changes in voltage or frequency. Recent work [10] seeks to relax this constraint, moving the granularity of scaling to the individual core. We expect processors in which each core can have its voltage and frequency adjusted independently. This form of heterogeneity is not only dynamic, but it will change frequently during execution.

Figure 2 summarizes the four sources of heterogeneity discussed in this section.

2.2 Impact of Heterogeneity

In this section, we use a simple experiment to demonstrate that smart OS scheduling of DHMPs can provide a great advantage—in terms of energy-efficiency and/or performance—over a scheduler that is unaware of the heterogeneity. The dynamic heterogeneity we consider in this experiment is due to faults that disable parts of cores.

In this experiment, which we presented in our previous work on fault diagnosis [5], we deconfigured portions of a single-core, SMT-enabled processor, similar to the Intel Pentium 4 [2]. Our hypothesis was that deconfiguration of one out of multiple units present in a core would result in a tolerable performance loss, making it favorable to seek a design that supports fault diagnosis and deconfiguration on a fine granularity. Data collected for that work is shown in Figure 3. Indeed, we were able to show that the loss of a single instance of a replicated

1. The performance heterogeneity introduced by process variability is a function of temperature, and thus there might be an additional dynamic aspect to it that we will not pursue here.

2. If the fault is in a singleton unit, such as a core's only floating point divider, then the core may not be salvageable. However, most components in a core are not singletons and we can thus tolerate deconfiguring them.

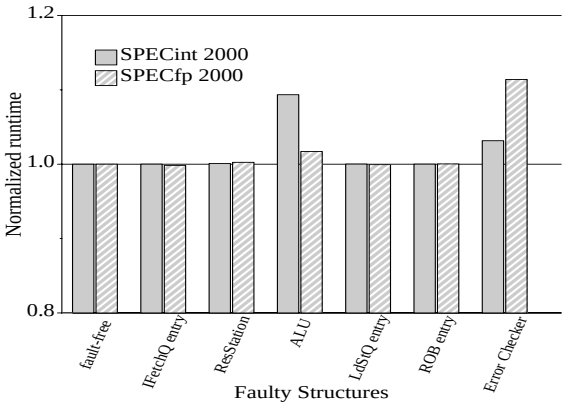

Figure 3. Performance (runtime) Impact of a Fault Causing the Loss of a Component in an SMT-Enabled Single-Core Processor.

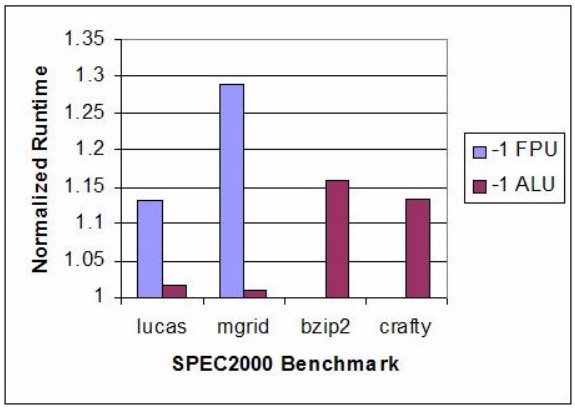

Figure 4. Performance (runtime) Impact of a Fault Causing the Loss of an ALU or FPU for Selected SPEC2000 Benchmarks.

functional unit results in a very small (<10%) loss in performance for a single-threaded SPEC2000 workload.

Figure 4 shows per-benchmark results for a subset of the data presented in Figure 3. In this figure, we observe that certain benchmarks are more sensitive to the loss of a particular functional unit. If we consider this data in the context of a DHMP, it shows that an intelligent scheduler could adapt to this heterogeneity to provide performance nearly equal to the fault-free scenario. Consider a 2-core processor in which Core1 has a faulty ALU and Core2 has a faulty FPU. If the scheduler knows to schedule mgrid on Core1 and bzip2 on Core2, then the runtime would be close to the fault-free case. However, if the scheduler obliviously schedules them the other way, then the runtime will suffer greatly. Even if the performance impact is not user-visible, the energy impact is still significant. When a program takes longer to run, it consumes more processor energy and can reduce opportunities for energy-saving optimizations such as disk spindowns.

This simple experiment shows that we want a scheduler that can dynamically migrate workloads to the cores that can support them best. The question is how do we achieve this goal?

3 Challenges for OS and Architecture

There are three fundamentally new challenges for efficiently scheduling DHMPs. First, the OS must discover the dynamic status of each core in order to know how much computational capability each core can currently supply. Second, the OS must discover the dynamic resource demand of each thread. Third, given the knowledge about core "supply" and thread "demand," the OS must match threads and cores as efficiently as possible.

3.1 Core Supply

In any heterogeneous multicore processor (static or dynamic), the OS scheduler will require basic information about the operational state of the present cores. For example, Core1 might be at its maximum supply (fully-functional and at its highest voltage and frequency), Core2 might have a faulty ALU, and Core3 might be at a lower frequency to save power. As we saw in our case study in Section 2.2, an OS that was not aware of heterogeneity in core supply (in that case, due to faults) may not schedule nearly as well as an OS with that knowledge.

For processors that rely on static scheduling of instructions, such as the Intel Itanium, the problems posed by dynamic heterogeneity are exacerbated. The compiler decides how to schedule the instructions, based on its (static) knowledge of the core's supply. If the core's supply changes, then this schedule may be obsolete and lead to degradations in power-efficiency and performance. Moreover, for statically scheduled cores with little or no hardware to adjust to run-time conditions, such as the Transmeta Crusoe [6], a change in the core supply can actually lead to incorrect execution. For multicore processors with cores like the Crusoe, we would want the OS to learn of supply changes in case re-compilation is preferable to moving the thread to a core with the expected supply.

Communicating core supply information to the OS will require support from the hardware. Architects will need to provide this information in the form of hardware performance counters, operational state descriptors, or explicit signals to the OS. Whatever the form of information exchange, it should be abstract enough to be uniform across a range of processor implementations.

3.2 Thread Demand

In a homogeneous multicore processor, differences in thread behavior generally do not matter to the scheduler. Some schedulers [9, 8] consider a thread's cache usage—whether the thread has warmed up its cache or is competing with another thread for a shared L2 cache—but they generally do not consider other aspects of the thread's behavior (e.g., whether it is memory bound, floating-point intensive, etc.). However, if cores are heterogeneous, then the scheduler would like to be aware of these differences in thread behavior. Our simple case study in Section 2.2 showed the importance of knowing the different resource demands of benchmarks such as mgrid and bzip. Moreover, each thread's behavior changes as it passes through phases of execution. Sherwood et al. [16] have studied programmatic phase behavior extensively, showing that demands on the underlying hardware change for periods on the order of ten million to hundreds of millions of instructions. Intuitively, in a DHMP, it will be desirable for a scheduler to react appropriately to changes in phase that negatively impact the performance of a given thread on a particular core.

Communicating thread demand to the OS could be performed by either the hardware, compiler, or some combination of the two. Hardware performance counters or meta-data from the compiler can be used to provide thread demand characteristics, but we need to determine what information to provide and when to provide it. If feasible, the scheduler wants the most amount of information at the greatest frequency. However, providing a rich and highly-dynamic set of demand data may be costly—in terms of hardware, performance, and power—and it may also over-burden the scheduler that tries to assimilate all of this information.

3.3 Scheduling

Once the OS and architecture communities define an interface for communicating supply and demand between the cores and the OS, the OS community must then develop scheduling algorithms that incorporate this information. Furthermore, these scheduling algorithms must also consider existing issues that are not related to dynamic heterogeneity, such as fairness, priority, real-time requirements, etc.

The design space for such scheduling algorithms is immense, and it is not even clear what are the most appropriate metrics for evaluating such algorithms. We are currently in the early stages of developing scheduling algorithms, and this process requires us to iterate with the development of supply and demand interfaces. We are also trying to exploit aspects of the vast amount of prior work in load balancing for traditional multi-chip multiprocessors and distributed systems.

4 Current State of the Art

There has been some preliminary work in scheduling for heterogeneous multicore processors, but it is far from solving all of the issues posed by dynamic heterogeneity. Ghiasi et al. [10] and Kumar et al. [14] constrained the heterogeneity to static configurations, such that the scheduler has a fixed, processor dependent knowledge of the underlying hardware capability. DeVuyst et al. [7] use a sampling interval to set scheduling policy for a fixed epoch of time. This algorithm will not scale well as the number of cores continues to increase. Further, it fails to recognize phase changes when they occur, which may lead to performance loss. Much of this prior work has been developed by architects who needed a functional, but not necessarily efficient, scheduler for purposes of evaluating their architectural ideas. These designs have been ad-hoc in nature, and there is significant opportunity for the OS community to apply their accumulated knowledge and experience to this problem.

Some other recent work has explored how heterogeneity impacts the OS. Balakrishnan et al. [1] observe that an OS scheduler that is aware of (static) core heterogeneity can, in some cases, overcome performance unpredictability caused by heterogeneity. Wells et al. [19] use a VMM-like layer to tolerate intermittent hardware faults by mapping multiple virtual cores to a single fault-free physical core.

For homogeneous multicore processors, Fedorova et al. [9, 8] have developed novel schedulers that consider the impact of L2 cache sharing among threads on the chip. This type of L2 sharing is unique to multicore chips, but it is orthogonal to the issue of dynamic heterogeneity.

5 Call to Action

Scheduling of tasks on DHMPs is a new problem. We believe that the OS community must develop schedulers that can handle DHMPs. Such schedulers will require dynamic knowledge of core status and thread demand. We will need to develop an interface between the hardware and the OS that enables the communication of this information. Because of these requirements, we expect that collaboration between the OS and architecture communities will be vital to achieving this goal.

References

[1] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The

Impact of Performance Asymmetry in Emerging Multicore Architectures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 506–517, June 2005.

[2] D. Boggs et al. The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology. *Intel Technology Journal*, 8(1), Feb. 2004.

[3] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16, Nov/Dec 2005.

[4] F. A. Bower, S. Ozev, and D. J. Sorin. Autonomic Microprocessor Execution via Self-Repairing Arrays. *IEEE Transactions on Dependable and Secure Computing*, 2(4):297–310, Oct-Dec. 2005.

[5] F. A. Bower, D. J. Sorin, and S. Ozev. A Mechanism for Online Diagnosis of Hard Faults in Microprocessors. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2005.

[6] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 15–24, Mar. 2003.

[7] M. DeVuyst, R. Kumar, and D. M. Tullsen. Exploiting Unbalanced Thread Scheduling for Energy and Performance on a CMP of SMT Processors. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, Apr. 2006.

[8] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance Of Multithreaded Chip Multiprocessors And Implications For Operating System Design. In *Proceedings of USENIX 2005 Annual Technical Conference*, Apr. 2005.

[9] A. Fedorova, M. Seltzer, and M. D. Smith. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2007.

[10] S. Ghiasi, T. Keller, and F. Rawson. Scheduling for Heterogeneous Processors in Server Systems. In *Proceedings of the 2nd Conference on Computing Frontiers*, May 2005.

[11] M. Gschwind et al. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, Mar/Apr 2006.

[12] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM POWER5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, Mar/Apr 2004.

[13] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, Mar/Apr 2005.

[14] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Exploiting Unbalanced Thread Scheduling for Energy and Performance on a CMP of SMT Processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2006.

[15] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous Chip Multiprocessors. *IEEE Computer*, pages 32–38, Nov. 2005.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[17] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2004.

[18] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[19] P. M. Wells, K. Chakraborty, and G. S. Sohi. Adapting to Intermittent Faults in Multicore Systems. In *Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2008.