The Impact of Dynamically Heterogeneous Multicore Processors on Thread Scheduling

ALTHOUGH MOST CURRENT MULTICORE PROCESSORS ARE HOMOGENEOUS, MICROARCHITECTS ARE NOW PROPOSING HETEROGENEOUS CORE IMPLEMENTATIONS, INCLUDING SYSTEMS IN WHICH HETEROGENEITY IS INTRODUCED AT RUNTIME. THIS ARTICLE SHOWS THAT OPERATING SYSTEM SCHEDULERS MUST CONSIDER DYNAMIC HETEROGENEITY OR SUFFER SIGNIFICANT POWER-EFFICIENCY AND PERFORMANCE LOSSES.

Fred A. Bower
IBM and Duke University

Daniel J. Sorin
Landon P. Cox
Duke University

•••••• Moore's law provides computer architects with more transistors than they can effectively use to extract instruction-level parallelism (ILP) in a single core. Thus, all current and future high-performance processor chips are multicore processors, also known as chip multiprocessors (CMPs). These multicore processors include the Cell Broadband Engine,[1] Intel's Core Duo and Quad-Core Xeon, AMD's Dual-Core Opteron, Sun Microsystems' Niagara,[2] and IBM's Power5.[3] 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 might not appear to present a substantially new problem for operating systems. There is a long history of OS scheduling for multithreaded microprocessors and traditional multichip multiprocessors. There has even been recent research into one aspect of scheduling that is unique to multicores, which is that processors often share L2 caches.[4,5] Aside from 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 that future multicore processors are unlikely to consist of homogeneous cores.[6,7] With core specialization, 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 runtime events such as physical faults or power management). In this article, 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

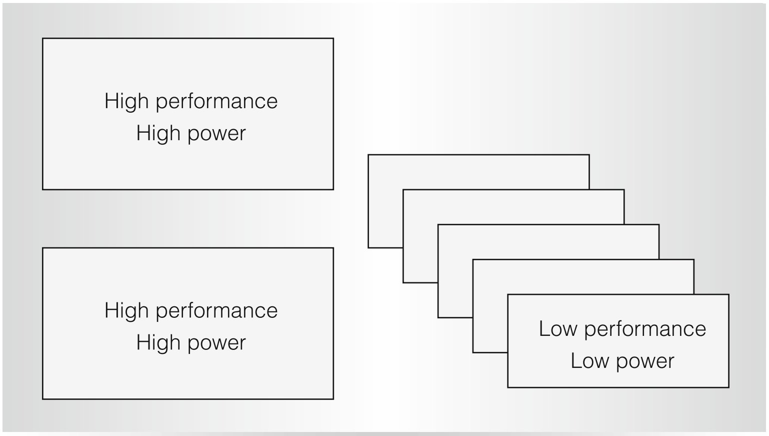

Figure 1. Statically heterogeneous multicore design: A few high-power, high-performance cores handle latency-sensitive tasks; several simpler, low-power cores execute throughput-oriented tasks.

acceptable power cost. A poor schedule will match each thread with a core that either cannot run it at an acceptable performance (for example, 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. Through two simple experiments, we frame the issues related to scheduling these systems; we also discuss the limited research that has been done in this area.

Multicore trends and impact

With the increasing transistor budgets afforded by Moore's law, architects have sought power-efficient ways to use all of the transistors they are allocated. Until recently, architects dedicated their transistor budget to extracting 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),[8] such as in the Pentium 4, in which multiple threads share a single core's resources. Unfortunately, a single SMT processor is also limited by how many transistors it can use power efficiently. As a result, the industry has begun placing multiple independent cores on each chip. Each of these cores might itself be multithreaded, providing a multiplicative number of schedulable contexts.

Homogeneous multicore processors consist of identical cores that provide a consistent computing capability for each schedulable context. Homogeneity simplifies the scheduler's job and enables the use of existing scheduling algorithms for multiprocessor systems. These algorithms might factor cache warmth and cache sharing into scheduling decisions, but they generally do not discriminate among cores, viewing all cores as equally capable of performing computations.

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) stresses the power and cooling limits for the chip. There is a fixed amount of power that a chip can consume before it becomes impossible to cool it (through air cooling). Given this power budget, a chip cannot contain dozens of Pentium 4-like processors, even if each one is itself power efficient. Nevertheless, for high-priority tasks, the single-threaded performance provided by current high-performance cores is still desirable. Thus, architects believe that (statically) heterogeneous multicore designs, such as the Cell processor,[1] will be prevalent in coming generations.[6,7] As Figure 1 shows, a multicore design might consist of a few high-power and high-performance cores, coupled with several simpler, low-power cores. These low-power cores will be tailored to execute the throughput-oriented tasks for which a user might tolerate greater latency.

In addition to static heterogeneity, we also expect dynamic heterogeneity because of at least three technological issues. Figure 2 summarizes these sources of heterogeneity: process variability, physical faults, and dynamic voltage and frequency scaling.

Process variability. Fabrication process variability is increasing,[9] and it is highly likely

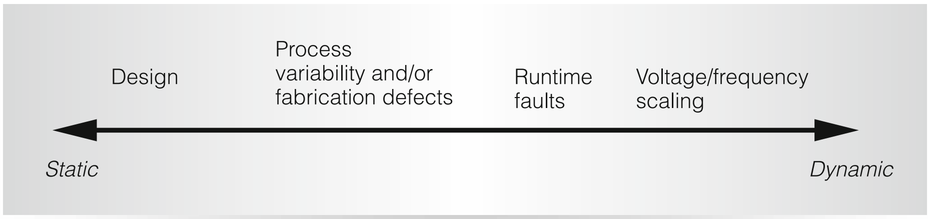
Design
Process variability and/or fabrication defects
Runtime faults
Voltage/frequency scaling
Static
Dynamic

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 can change every scheduling quantum, requiring scheduler adaptability to effectively exploit the heterogeneity.

that the cores will have different performance characteristics. Thus, it will be preferable for each core to have a different maximum frequency rather than derate the entire chip to the lowest-common maximum 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. (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.)

By decreasing the maximum frequency of a given core, we impose a performance degradation on it that is fairly uniform, in terms of affecting all benchmarks similarly. When decreasing frequency, the only nonuniformity in performance degradation is due to the relative decrease in memory access latency. As a result, instructions per cycle (IPC) could increase slightly and lead to an overall slowdown that is less than linear as a function of frequency.

Physical faults. As CMOS trends continue to lead toward smaller device and wire dimensions, the probability of hard (permanent) faults in microprocessors increases. These faults can be introduced during chip fabrication or in the field. Well-known physical phenomena that lead to operational hard faults are gate oxide breakdown, electromigration, and thermal cycling. Microprocessors become more susceptible to all of these phenomena as device dimensions shrink,[10] and the semiconductor industry's roadmap has identified hard faults as critical challenges (http://www.itrs.net). Blome et al.[11] recently analyzed the MTTF (mean time to failure) of the OpenRISC 1200 core in 90-nm technology,[12] and they discovered that hard faults are likely to occur during the core's lifetime. In the near future, with even smaller CMOS technologies, it might no longer be a cost-effective strategy to discard a core with one or more hard faults, which is what commonly occurs today.

Our prior research has explored how to detect and diagnose permanent faults in a single superscalar core.[13,14] In response to these faults, part of the core might be deconfigured, resulting in the core moving to a lower-performing state, but still providing useful work to the system. (If the fault is in a singleton unit, such as a core's only floating-point divider, then the core might not be salvageable. However, most of a core's components are not singletons, and we can thus tolerate deconfiguring them.) The performance degradation is nonuniform across benchmarks, because benchmarks are more or less sensitive to various core features. For example, a benchmark that is memory bound might not incur much performance penalty if it runs on a core that has a fault in one of its three ALUs.

Deconfiguration due to hard faults is a fairly rare event, and it can occur either at manufacturing time (to address fabrication defects) or during the part's lifetime (to address wearout faults). If we apply this approach or a similar technique to a multicore chip, even one that is designed to be homogeneous, it will lead to a DHMP.

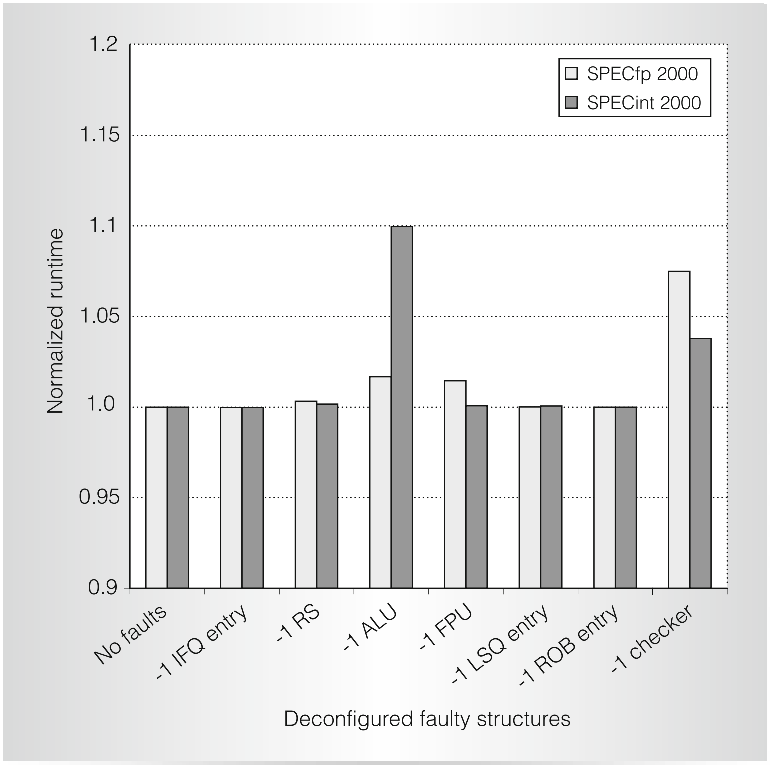

Figure 3. Performance (runtime) impact of a fault causing the loss of a component in a single-core processor with simultaneous multithreading (SMT).

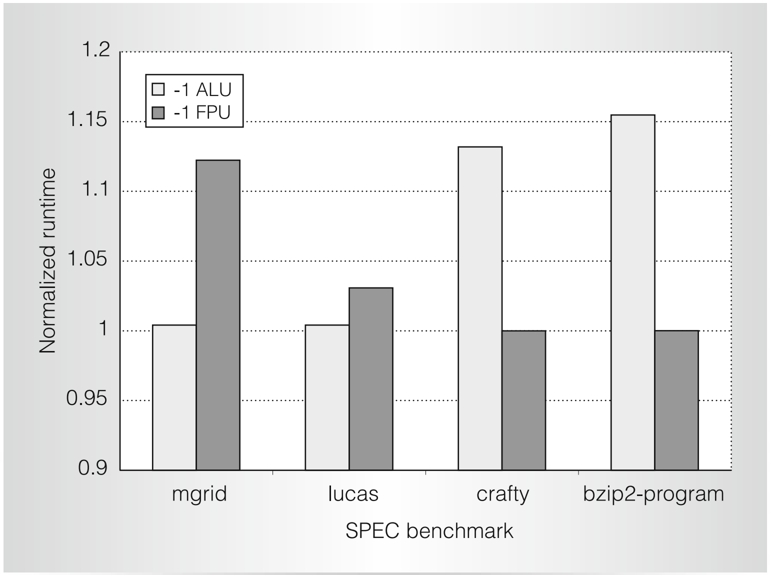

Figure 4. Performance (runtime) impact of a fault causing the loss of an ALU or FPU for selected SPEC 2000 benchmarks.

Dynamic voltage and frequency scaling. Each core is likely to incorporate its own dynamic voltage and frequency scaling. DVFS techniques are in use today, but are constrained to chip-wide changes in voltage or frequency. Recent work seeks to relax this constraint, moving the granularity of scaling to the individual core.[15–17] We expect processors in which each core can have its voltage and frequency adjusted independently. This form of heterogeneity is dynamic and will change frequently during execution.

Impact of heterogeneity

Here, we use two simple experiments to demonstrate that smart OS scheduling of DHMPs can provide a great advantage—in terms of performance and energy-efficiency—over a scheduler that is unaware of the heterogeneity.

Experiment 1. 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,[14] we deconfigured portions of a single-core, SMT-enabled processor, similar to the Intel Pentium 4.[18] 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 at a fine granularity. Figure 3 shows data collected for that work. Indeed, we showed that the loss of a single instance of a replicated unit results in a small (less than 10 percent) loss in performance for a single-threaded SPEC CPU 2000 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 unit—an arithmetic logic unit (ALU) or floating-point unit (FPU) in this experiment. 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 two-core processor in which core 1 has a

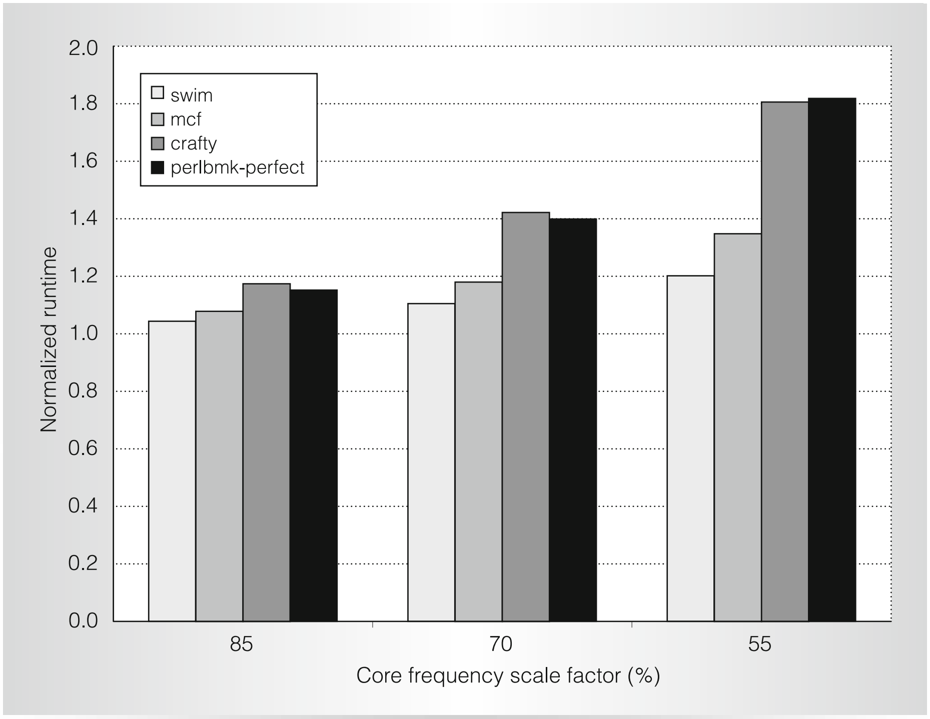

Figure 5. Performance (runtime) impact of dynamic frequency scaling.

faulty ALU and core 2 has a faulty FPU. If the scheduler knows to schedule mgrid on core 1 and bzip2 on core 2, 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 visible to the user, 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 spin-downs.

Experiment 2. In this experiment, we consider dynamic heterogeneity due to core frequency scaling. In Figure 5, we show the results of scaling the frequency while running four benchmarks. As with the previous experiment, in this one we observe a wide range of sensitivity across the benchmarks. For example, when the core's clock is scaled down to 55 percent of its maximum frequency, swim's runtime increases by 20 percent (with respect to a core at maximum frequency) while crafty and perl incur runtime increases in excess of 80 percent.

Although we show only a subset of the SPEC CPU2000 benchmarks in Figure 5, the data shows that benchmarks fall into three clusters in terms of their performance sensitivity to clock frequency. Benchmarks in the high-sensitivity cluster, which includes crafty and perlbmk, are nearly linear in their sensitivity to clock frequency scaling. The low-sensitivity cluster, which includes swim and mcf, shows sublinear performance degradation as frequency is decreased. The sublinearity arises because decreasing the frequency reduces the memory latency (in terms of number of cycles), and these benchmarks obtain an IPC benefit from this effect. The medium-sensitivity cluster falls between the first two clusters. When we classified data from our experiments into these three clusters, we found that the high-sensitivity cluster is largest, with roughly 56 percent of the benchmarks falling into this category. Somewhat to our surprise, the low-sensitivity cluster included 28 percent of the benchmarks. The remaining 16 percent of the benchmarks fell into the medium-sensitivity cluster. This data suggests an opportunity for an intelligent scheduler to be able to make better power-performance decisions.

Consider a two-core chip in which the scheduler must reduce the frequency of one core to avoid exceeding its power budget. Assume the chip wishes to run swim and crafty and that they have equal priorities. An oblivious scheduler could place crafty on the slower core, which would provide far worse performance than if swim were placed on the slower core. Once again, an intelligent scheduler could improve performance and save energy.

Conclusion. These two simple experiments show 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?

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 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.

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, core 1 might be at its maximum supply (fully functional and at its highest voltage and frequency), core 2 might have a faulty ALU, and core 3 might be at a lower frequency to save power. As we saw in our experiments, an OS that was not aware of heterogeneity in core supply might 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 on the basis of its (static) knowledge of the core's supply. If the core's supply changes, then this schedule might be obsolete and lead to degradations in power efficiency and performance. Moreover, for statically scheduled cores with little or no hardware to adjust to runtime conditions, such as Transmeta's Crusoe,[19] 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 recompilation 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.

Concretely, we expect each core to export a subset of the following information to the OS, along with perhaps other core supply metrics:

- maximum and current frequency,
- current voltage, and
- effective pipeline width for both integer and floating-point operations.

Thread demand

In a homogeneous multicore processor, differences in thread behavior generally do not matter to the scheduler. Some schedulers[4,5] 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 (such as whether it is memory bound, floating-point intensive, and so on). However, if cores are heterogeneous, the scheduler should be aware of these differences in thread behavior. Our simple experiments show the importance of knowing the different resource demands of different applications.

Moreover, each thread's behavior changes as it passes through phases of execution.

Sherwood et al. 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.[20] 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, the compiler, or some combination of the two. Hardware performance counters or metadata from the compiler can provide thread demand characteristics, but we need to determine what information is needed and when to deliver it. If feasible, the scheduler should have the most amount of information at the greatest frequency. However, providing a rich and highly dynamic set of demand data could be costly—in terms of hardware, performance, and power—and it could also overburden the scheduler that tries to assimilate all of this information.

In practice, we expect thread demand data to encompass a subset of the following metrics:

- available integer and floating-point ILP, and whether the thread is computing bound;
- memory demand and whether the thread is bound by L1 data cache bandwidth;
- the thread's sensitivity to L2 cache latency; and
- whether the thread is fetch bound.

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, and real-time requirements.

The design space for such scheduling algorithms is immense, and it is not even clear what the most appropriate metrics for evaluating such algorithms are. 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 multichip multiprocessors and distributed systems.

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.[15] and Kumar et al.[21] constrained the heterogeneity to static configurations, such that the scheduler has a fixed, processor-dependent knowledge of the underlying hardware capability. DeVuyst et al.[22] 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 was 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.[6] 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.[23] use a hypervisor-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.[4,5] 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.

We hope this article will serve as a call to action for research in scheduling DHMPs. Scheduling of tasks on DHMPs is a new problem, and 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. MICRO

References

1. M. Gschwind et al., ''Synergistic Processing in Cell's Multicore Architecture,'' *IEEE Micro*, vol. 26, no. 2, Mar.-Apr. 2006, pp. 10-24.
2. P. Kongetira, K. Aingaran, and K. Olukotun, ''Niagara: A 32-way Multithreaded SPARC Processor,'' *IEEE Micro*, vol. 25, no. 2, Mar.-Apr. 2005, pp. 21-29.
3. R. Kalla, B. Sinharoy, and J.M. Tendler, ''IBM POWER5 Chip: A Dual-Core Multithreaded Processor,'' *IEEE Micro*, vol. 24, no. 2, Mar.-Apr. 2004, pp. 40-47.
4. A. Fedorova et al., ''Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design,'' *Proc. Usenix 2005 Ann. Technical Conf.*, Usenix Assoc, 2005, p. 26.
5. A. Fedorova, M. Seltzer, and M.D. Smith, ''Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler,'' *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 07), 2007, pp. 25-38.
6. S. Balakrishnan et al., ''The Impact of Performance Asymmetry in Emerging Multicore Architectures,'' *Proc. 32nd Ann. Int'l Symp. Computer Architecture* (ISCA 05), IEEE CS Press, 2005, pp. 506-517.
7. R. Kumar et al., ''Heterogeneous Chip Multiprocessors,'' *Computer*, vol. 38, no. 11, Nov. 2005, pp. 32-38.
8. D.M. Tullsen et al., ''Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor,'' *Proc. 23rd Ann. Int'l Symp. Computer Architecture* (ISCA 96), IEEE CS Press, 1996, pp. 191-202.
9. S. Borkar, ''Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation,'' *IEEE Micro*, vol. 25, no. 6, Nov.-Dec. 2005, pp. 10-16.
10. J. Srinivasan et al., ''The Impact of Technology Scaling on Lifetime Reliability,'' *Proc. Int'l Conf. Dependable Systems and Networks* (DSN 04), IEEE CS Press, 2004, p. 177.
11. J. Blome et al., ''Self-Calibrating Online Wearout Detection,'' *Proc. 40th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 07), IEEE CS Press, 2007, pp. 109-122.
12. D. Lampret, *OpenRISC 1200 IP Core Specification*, Rev. 0.7 2001http://www.opencores.org.
13. F.A. Bower, S. Ozev, and D.J. Sorin, ''Autonomic Microprocessor Execution via Self-Repairing Arrays,'' *IEEE Trans. Dependable and Secure Computing*, vol. 2, no. 4, Oct-Dec. 2005, pp. 297-310.
14. F.A. Bower, D.J. Sorin, and S. Ozev, ''A Mechanism for Online Diagnosis of Hard Faults in Microprocessors,'' *Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 05), IEEE CS Press, 2005, pp. 197-208.
15. S. Ghiasi, T. Keller, and F. Rawson, ''Scheduling for Heterogeneous Processors in Server Systems,'' *Proc. 2nd Conf. Computing Frontiers* (CF 05), ACM Press, 2005, pp. 199-210.
16. C. Isci et al., ''An Analysis of Efficient Multicore Global Power Management Policies: Maximizing Performance for a Given Power Budget,'' *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 06), IEEE CS Press, 2006, pp. 347-358.
17. J. Sartori and R. Kumar, *Proactive Peak Power Management for Many-Core Architectures*, tech. report CRHC-07-04, Center for Reliable and High-Performance Computing, Univ. of Illinois at Urbana-Champaign, 2007.
18. D. Boggs et al., ''The Microarchitecture of the Intel Pentium 4 Processor on 90 nm Technology,'' *Intel Tech. J.*, vol. 8, no. 1, Feb. 2004, pp. 1-18.
19. J.C. Dehnert et al., ''The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges,'' *Proc. Int'l*

Symp. Code Generation and Optimization (CGO 03), IEEE CS Press, 2003, pp. 15-24.

20. T. Sherwood et al., ''Automatically Characterizing Large Scale Program Behavior,'' *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 45-57.
21. R. Kumar, D.M. Tullsen, and N.P. Jouppi, ''Core Architecture Optimization for Heterogeneous Chip Multiprocessors,'' *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 06), ACM Press, 2006, pp. 23-32.
22. M. DeVuyst, R. Kumar, and D.M. Tullsen, ''Exploiting Unbalanced Thread Scheduling for Energy and Performance on a CMP of SMT Processors,'' *Proc. IEEE Int'l Parallel and Distributed Processing Symposium* (IPDPS 06), 2006, p. 117.
23. P.M. Wells, K. Chakraborty, and G.S. Sohi, ''Adapting to Intermittent Faults in Multicore Systems,'' *Proc. 13th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 08), ACM Press, 2008, pp. 255-264.

Fred A. Bower is a senior engineer in the Systems and Technology Group at IBM, and is pursuing his PhD in computer science at Duke University. His research interests focus on processor microarchitecture and the exposed interface to system software. He has a BS in mechanical engineering and computer science from Oregon State University and an MS in computer science and engineering from the Oregon Graduate Institute.

Daniel J. Sorin is an assistant professor of electrical and computer engineering and of computer science at Duke University. His research interests are centered on computer architecture, with a focus on dependability and verification. He has a BSE in electrical engineering from Duke University, and an MS and a PhD in electrical and computer engineering from the University of Wisconsin.

Landon P. Cox is an assistant professor in the Computer Science Department at Duke University. His research interests include distributed systems and operating systems. He has a BS in computer science from Duke University, and an MS and a PhD in computer science and engineering from the University of Michigan.

Direct questions and comments about this article to Daniel J. Sorin, PO Box 90291, Durham, NC 27708; sorin@ee.duke.edu.