Thread-Management Techniques to Maximize Efficiency in Multicore and Simultaneous Multithreaded Microprocessors

R. RAKVIC

United States Naval Academy

and

Q. CAI, J. GONZÁLEZ, G. MAGKLIS, P. CHAPARRO, and A. GONZÁLEZ

Intel Labs-UPC

9

We provide an analysis of thread-management techniques that increase performance or reduce energy in multicore and Simultaneous Multithreaded (SMT) cores. Thread delaying reduces energy consumption by running the core containing the critical thread at maximum frequency while scaling down the frequency and voltage of the cores containing noncritical threads. In this article, we provide an insightful breakdown of thread delaying on a simulated multi-core microprocessor. Thread balancing improves overall performance by giving higher priority to the critical thread in the issue queue of an SMT core. We provide a detailed breakdown of performance results for thread-balancing, identifying performance benefits and limitations. For those benchmarks where a performance benefit is not possible, we introduce a novel thread-balancing mechanism on an SMT core that can reduce energy consumption. We have performed a detailed study on an Intel microprocessor simulator running parallel applications. Thread delaying can reduce energy consumption by 4% to 44% with negligible performance loss. Thread balancing can increase performance by 20% or can reduce energy consumption by 23%.

Categories and Subject Descriptors: C.1.2 [Computer Systems Organization]:—multiple Data Stream Architectures—parallel processors

General Terms: Design, Performance

Additional Key Words and Phrases: Meeting point thread characterization, critical threads, thread delaying, thread balancing, multi-threaded application, microarchitecture, low-power, energy-aware

Author's addresses: R. Rakvic, United States Naval Academy, Annapolis, MD; Q. Cai, J. González, G. Magklis, P. Chaparro, A. González, Intel Labs-UPC, Barcelona, Spain; email: rakvic@usna.edu. © 2010 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the [U.S.] Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org. © 2010 ACM 1544-3566/2010/04-ART9 \$10.00

DOI 10.1145/1839667.1839671 <http://doi.acm.org/10.1145/1839667.1839671>

ACM Transactions on Architecture and Code Optimization, Vol. 7, No. 2, Article 9, Publ. date: September 2010.

ACM Reference Format:

Rakvic, R., Cai, Q., González, J., Magklis, G., Chaparro, P. and González, A. 2010. Thread-management techniques to maximize efficiency in multicore and simultaneous multithreaded microprocessors. *ACM Trans. Architect. Code Optim.* 7, 2, Article 9 (September 2010), 25 pages. DOI = 10.1145/1839667.1839671 <http://doi.acm.org/10.1145/1839667.1839671>

1. INTRODUCTION

It has now become apparent that Chip Multiprocessors (CMPs) [Olukton et al. 1996] have become a common way to extract performance in the microprocessor industry [Borkar 2005; Intel 2006; Kongetira et al. 2005]. In addition, some processor companies, such as Intel Corporation, continue to employ Simultaneous Multithreaded (SMT) for the individual cores. Some believe that these microprocessors will be a match for the future applications, such as Recognition, Mining, and Synthesis [Intel 2005], which may be highly parallel.

In addition to the goal of high performance, low-energy consumption will be one of the major challenges in the design of such systems. The workload imbalance among cores in a CMP chip is one source of energy inefficiency. For example, in a fork-join parallel execution model, such as OpenMP [2005], a parallel loop usually has a barrier at the join point of the loop that synchronizes all threads. In the best case, all cores reach this barrier at the same time. However, in a normal situation, some threads reach the barrier earlier than others and spend a large amount of time waiting for slower ones. Fast threads have been executed at the maximum possible speed and power consumption, which leads to energy inefficiency.

We utilize a mechanism called *meeting point thread characterization* [Cai et al. 2008] that identifies the critical thread of a single multithreaded application as well as the amount of slack of noncritical threads. To do that, each thread has a counter to accumulate the number of iterations executed for the parallel loop. At specific intervals of time, all threads broadcast this information so they can know the number of iterations being executed by each one of them. With that information, the slack of a thread can be estimated as the difference between its own iteration counter and the counter of the slowest one. We believe that the meeting point mechanism is a powerful tool that enables many interesting optimizations. We focus our analysis on two of such optimizations that dynamically adapt the hardware resources to the application behavior: thread delaying and thread balancing [Cai et al. 2008], as well as a novel thread-balancing proposal that reduces energy.

The goal of thread delaying is to reduce overall energy consumption by dynamically scaling down the voltage and the frequency of the cores executing noncritical threads. At specific intervals of time, each core utilizes meeting point thread characterization to estimate the slack of the parallel thread. Then, it computes the voltage/frequency for the next interval of time so that the energy is minimized but the expected arrival time to the barrier does not exceed that of the current critical thread.

Thread balancing is a hardware scheme that works for simultaneous multithreading processors running parallel threads. The goal of thread balancing is to reduce the overall execution time by speeding up the critical thread. To

do that, the critical thread is given priority in the utilization of the issue slots. This approach is radically different from the issue policies already proposed in the literature [Burd and Brodersen 1995; El-Moursy and Albonesi 2003; Homayoun et al. 2005; Jain et al. 2002; Robatmili et al. 2004; Tullsen et al. 1996]. Previous works assume that the threads are from different applications, and the proposed issue algorithms try to maximize bandwidth utilization as well as fairness. However, our approach is completely different because threads come from the same parallel application. The only way to improve overall performance is to accelerate the critical thread. Therefore, in our approach, higher priority is given to the critical thread.

In addition, for those applications where performance is not garnered, we propose a novel energy-savings technique via thread balancing. In this approach, instead of prioritizing issue bandwidth, clockgating will be employed upon a load miss. During this time, both threads will be turned off. As a result, energy will only be consumed during times when the slow, or critical thread is making forward progress. The overall number of energy consuming cycles will be reduced.

We have evaluated thread delaying and thread balancing in cycle-accurate CMP and SMT simulators, respectively. Our experiments with several Recognition, Mining, and Synthesis (RMS) workloads show that thread delaying on a CMP system can greatly reduce energy (from 4% to 44%) with negligible performance penalty. In this article, we also show individual benchmark results highlighting the runtime behavior of our mechanisms.

The experiments on an SMT in-order core show that our thread balancing mechanism can improve performance for various RMS workloads up to 20%. In this article, we also provide a detailed breakdown of performance results for thread balancing, identifying performance benefits and limitations. We illustrate causes of stalls, levels of imbalance, and reasons for performance limitations. For those benchmarks where a performance benefit is not possible, we propose an energy-saving technique on an SMT core that can save up to 23% of energy consumption while only losing 3% of performance.

The rest of this article is organized as follows. We first describe the meeting point mechanism in Section 2. The thread-delaying and thread-balancing techniques are explained in detail in Sections 3 and 4, respectively. Section 5 shows the performance results and detailed analysis of thread delaying and thread balancing. Related work is discussed in Section 6, and a conclusion is provided in Section 7.

2. IDENTIFICATION OF CRITICAL THREADS

The meeting point thread characterization aims at detecting dynamically the workload imbalance of parallel applications. Figure 1 demonstrates that even very regular parallel programs may exhibit workload imbalance during execution. Figure 1(a) shows the main parallel loop from *PageRank-lz77* (a RMS workload). The code is already written in such a way that the input data set is partitioned to achieve workload balance. However, Figure 1(b) shows that workload imbalance still exists on a two-core system. The x -axis in Figure 1(b)

Fig. 1. A motivational example. (a) A parallelized loop from PageRank (lz77 method). (b) Performance results for two cores. (c) Insertion of a meeting point.

represents the number of iterations of the outermost loop that each core executes. The y-axis represents the cumulative execution time of this parallel loop for each core. We can see that Core 1 is slower than Core 0. In this particular case, the reason is that Core 1 suffers many more cache misses than Core 0 does. Other reasons for workload imbalance could be that parallel threads follow different control paths in the parallel loop, or that the application exploits task-level parallel, rather than loop level. We refer to this slow thread as the critical thread because the other threads must wait for it due to the barrier at the end of the parallel section.

We propose to identify the critical thread dynamically during program execution by checking the workload balance at intermediate points of a parallel loop. We call these checkpoints meeting points. A natural location of a meeting point is at the back edge of a parallel loop, because the back edge of a loop is visited many times by all threads at runtime. It should be noted that the total number of times each thread visits the meeting point should be roughly the same, which means that the total amount of work assigned to each thread should be the same. Otherwise, the critical thread cannot be identified based on the number of times the threads visit a meeting point. In the case of the OpenMP programming model [2005], this assumption is usually true if static scheduling is applied.

In the OpenMP programming model, if parallel codes are extremely irregular, dynamic scheduling can be used. Our critical thread identification is not suitable for this scenario. However, dynamic scheduling has large runtime overheads and static scheduling is recommended as the first scheduling option, especially when the number of threads is increased [Zhu et al. 2006]. The decision whether to use static or dynamic scheduling in a parallel is out of the scope of this article.

The process of our meeting point thread characterization normally consists of the following three steps.

(1) *Insertion of meeting points.* One candidate for a meeting point is the place in a parallel region that is visited by all threads many times during parallel execution. For example, in Figure 1(c), we have a program using the parallel for

construct of the OpenMP programming model. As the code is regular, it is easy to see that the backward branch of the outermost loop satisfies our criteria.

The insertion of a meeting point can be done by the hardware, the compiler, or the programmer. Although a hardware-only approach is completely transparent and maintains binary compatibility, it requires extra hardware structure to detect a suitable meeting point among repeated instructions in a parallel execution. Hardware schemes for backward loop detection could be used [Marcuello et al. 1998].

(2) *Identification of critical threads.* Every time a core decodes an instruction encoding a meeting point, a thread-private counter is incremented. This counter is a proxy for the aforementioned slack. The most critical thread is the one with the smallest counter, and the slack of a thread can be estimated as the difference of its counter and the counter of the slowest counter.

Depending on the usage of our meeting point thread characterization, a software-only identification mechanism could be adopted. For example, the application is rewritten so that it includes an array of counters indexed by thread identifiers. Each thread increments its own counter every time it arrives at the end of the parallel section.

In this work, the user inserts the meeting point by means of a pragma and the counters are implemented in hardware. The compiler translates the pragma into a new instruction that, once decoded, increments the private hardware counter of the thread.

(3) *Usage of criticality information.* The usage of thread criticality or slack estimation depends on what optimizations we want to apply. For example, we will demonstrate two applications in later sections. One, called thread delaying, minimizes energy consumption by slowing down the fast threads. The other application, called thread balancing, optimizes performance by accelerating the slowest thread.

3. THREAD DELAYING

As discussed in Section 2, parallel applications exhibit workload imbalance among threads at runtime. In a fork-join parallel-programming model such as OpenMP, workload imbalance means that noncritical threads finish their jobs earlier than their critical counterparts do. Since there is a barrier at the join point of a parallelized loop, noncritical threads will have to wait for the critical thread to finish its work before they can proceed. In modern systems, the CPUs of the noncritical threads can be put into deep sleep mode, which consumes almost zero energy [Fischer 2007]. However, this is not the most energy-efficient approach to deal with workload imbalance. Due to the cubic relationship of power to frequency/voltage, it is better to make noncritical threads run at a lower frequency/voltage level such that all threads arrive at the barrier at the same time.

3.1 Energy Savings due to DVFS

Assume that the critical thread finishes its work in T time units, and a noncritical thread can finish its work in only $0.7T$ time units. If the noncritical thread

Fig. 2. (a) Our CMP microarchitecture. (b) Voltage-frequency table. (c) Two tables are required to implement thread delaying.

works at full speed for $0.7T$ time units and then it is put to deep sleep mode with zero energy consumption for the rest $0.3T$ time units, the total consumption from this noncritical thread is given by the following formula.

$$E_{f_{\max}} = c \times V_{dd}^2 \times f_{\max} \times 0.7T.$$

Alternatively, the core running the noncritical thread can have its frequency scaled down to $0.7 f_{\max}$ and it would meet the barrier on time anyway. In this case, the total consumption for the noncritical thread is as follows.

$$E_{f_{\text{scaled}}} = c \times (0.875V_{dd})^2 \times V_{dd}^2 \times (0.7 f_{\max}) \times T = 0.765E_{f_{\max}}.$$

From the previously described deductions, we can clearly see the advantage of doing DVFS on noncritical threads. There are two main challenges by applying DVFS in this scenario. First, we need a way to identify noncritical and critical threads at runtime. Second, we need to select appropriate frequency and voltage levels for noncritical threads. In this section, we will describe a new algorithm, called *thread delaying*, which solves these two problems by combing the meeting point thread characterization technique and an estimation formula for predicting the frequency/voltage levels for each thread.

3.2 A CMP Microarchitecture with Multiple Clock Domains

Figure 2(a) shows the baseline of our CMP microarchitecture. Our CMP processor consists of many Intel64/IA32 cores. Each core, due to power and temperature constraints, is a single-threaded in-order core with bandwidth of two instructions per cycle. Every core contains a private first-level instruction cache, a private first-level data cache and a private second-level unified cache. A shared third-level cache (L3) is connected to all cores through a bus network. A MESI cache protocol is used to keep data coherent.

Each core with associated L1 and L2 caches belongs to a separate clock domain. Moreover, the unified L3 cache with the interconnect forms a separate clock domain as well. Each clock domain has its own local clock network that receives as input a reference clock signal and distributes it to all the circuits of the domain. In our design, we assume that the phase relationship (i.e., the skew) between the domain reference clocks can be arbitrary. First, this allows

to run each domain at a different frequency. Second, to adapt the frequency of each domain dynamically and independently of the others. Since domains operate asynchronously to each other, interdomain communication must be synchronized correctly to avoid metastability [Chaney and Molnar 1973]. We use the mixed-clock FIFO design of Chelcea and Nowick [2001] to communicate values safely between domains.

Each one of the microprocessor domains can operate at a distinct voltage and frequency. Moreover, voltage and frequency can be changed dynamically and independently for each domain. We assume domains can execute through voltage changes, similar to previous studies [Iyer and Marculescu 2002; Magklis et al. 2004; Semeraro et al. 2004; Wu et al. 2005] and some commercial designs [Gochman et al. 2003]. We assume a limited range of voltages and frequencies, as shown in Figure 2(b).

Having so few levels allows us to switch between them very quickly. We assume a single, external PLL for the whole chip. Each domain includes an on-chip digital clock multiplier connected to the external PLL [Fischer et al. 2006; Olsson et al. 2000]. Frequency changes per domain are effected by changing the multiplication factor of the domain clock multiplier; the external PLL frequency is fixed. This allows extremely fast frequency changes, but it also means that (i) only a few frequency levels are available and (ii) all frequencies must be multiples of a base frequency.

3.3 Implementation of Thread Delaying

In order to implement thread delaying, each core contains two tables (shown in Figure 2(c)) to handle meeting points.

- MP-COUNTER-TABLE has as many entries as number of cores in the processor. Each entry contains a 32-bit counter that keeps track of the number of times each core has reached the given meeting point. This table is consistent among all cores in the system.
- HISTORY-TABLE includes an entry for each possible frequency level. Each entry contains a 2-bit up-down saturating counter used to determine the next frequency the core must run at. The table is initialized so that the entry corresponding to the maximum frequency level has the highest value. (i.e., all cores start running at maximum frequency.)

When a core decodes a meeting point, the counter corresponding to its assigned thread in the MP-COUNTER-TABLE is incremented by 1. Every 10 executions of the meeting point instruction, the core broadcasts the value of the counter to the rest of the cores. (Ideally, one would like to broadcast that information at every meeting point visit; however, the interconnection may be overloaded.) This is done by means of a special network message. When the network interface of a core receives such message, the MP-COUNTER-TABLE is accessed to increment by 10 the counter associated with the thread identifier of the sender. We choose 10, since it gives enough precision to the thread delaying with negligible impact on the interconnect performance.

Each core manages its own frequency and voltage independently, based on the value of the counter associated to its local thread in the MP-COUNTER-TABLE and the lowest value of all counters in the table. This corresponds to the critical thread, since it has executed the lowest number of iterations of the parallel loop. Therefore, we can say that the difference between both counters is an estimation of the slack of a thread.

Every 10 executions of the meeting point instruction, the processor frontend stops fetching instructions and inserts a microcode (stored in a local ROM) to execute the thread-delaying control algorithm. This microcode has dozens of instructions and its overhead has negligible impact on final performance. That microcode has as input both the MP-COUNTER-TABLE and the HISTORY-TABLE and its output is the frequency f for the next interval.

The microcode first computes the frequency that better matches the current slack, using the following formula:

$$f_{temp} = \frac{C_{critical}}{C_i} \times MAX_FREQUENCY$$

$$f_i = search_closest_valid_frequency(f_{temp})$$

where $C_{critical}$ and C_i are the counters from the critical thread and noncritical thread i , respectively. After f_{temp} is obtained, f_i is calculated by finding the minimum frequency supported in the system, whose value is equal or greater to f_{temp} . In our current model, voltage scaling is not implemented as a continuous function but a discrete one with 13 frequency levels [Chaparro et al. 2007; Fischer et al. 2006; Magklis et al. 2006; Olsson et al. 2000].

Once the frequency level for f_i is obtained, the HISTORY-TABLE is updated properly whereas each entry contains a 2-bit up-down saturating counter. If the frequency level for f_i is k , entry k is incremented and every other entry is decremented. Finally, the frequency chosen by the microcode for the next interval is the one with the largest counter in the HISTORY-TABLE.

Note that the purpose of HISTORY-TABLE is used to reduce the effect of temporal noise in the estimation of the slack, which may drive to the utilization of frequencies that are too aggressive (too low). This may cause a noncritical thread to become a critical one.

We have adopted the solution of inserting microcode in the processor to compute the next frequency, since this computation is not done very often and the overall performance is not affected. If this computation is critical, it could be done by pure hardware by adding the required functional units and control in the processor frontend. However, it is very difficult to justify the area increase to perform just this task and nothing else.

4. THREAD BALANCING

In Section 3, we have described a method to reduce energy consumption by slowing down noncritical threads on a CMP. In this section, we first focus on speeding up via thread balancing a parallel application running more than one thread on a single two-way SMT core by accelerating the critical thread. We also will present a technique that focuses on energy reduction with this technique.

Fig. 3. Balancing threads.

In an SMT core, the issue bandwidth is limited and shared among threads. There are a lot of issue policies in the literature [El-Moursy and Albonesi 2003; Homayoun et al. 2005; Jain et al. 2002; Robatmili et al. 2004; Tullsen et al. 1996], most of which assume that threads come from different applications (multiprogrammed workloads). A typical SMT microprocessor issue logic works as follows: If both threads have ready instructions, each one of them is allowed to issue one instruction. If one thread has ready instructions and the other does not, the one with ready instructions can issue up to two per cycle. A typical SMT tries to maximize bandwidth and fairness. However, if both threads belong to the same parallel application, fairness may not be the best option. After all, what we want is to speed up the parallel application and not a single thread. In this case, it is very important to identify the critical thread and give to it more priority in the issue logic—that is the purpose of our thread-balancing mechanism.

As stated previously, typically there is a barrier at the end of a parallelized loop that synchronizes the threads. In the ideal case, all threads reach this barrier at the same time. However, in a common situation, some threads reach this barrier earlier than others and spend a large amount of time waiting for slower threads to arrive; see Thread Imbalance in Figure 3. On a multithreaded core executing threads from the same application, a solution to correct this imbalance is to increase the priority of the slower threads so as to balance thread execution time (see Balanced Threads in Figure 3).

In this work, we present a novel mechanism called multibalancing threads that dynamically speeds up the slowest thread by increasing its priority to the issue bandwidth. On an in-order multithreaded core, the issue bandwidth is often limited, and the resource is shared amongst the threads. Figure 4 illustrates our multibalancing proposal at a conceptual level. Since more issue priority is given to the slowest thread, overall performance can be improved, resulting in balanced, and hence optimal, behavior. Our algorithm, which is described in the following text, consists of two simple-state machines that first analyze program execution imbalance and then prioritize issue bandwidth appropriately.

Fig. 4. Issue prioritization.

Fig. 5. Thread-balancing hardware algorithm.

Figure 5 provides an illustration of two key hardware steps required to perform multibalancing threads. First, logic is provided that identifies thread imbalance. Second, logic is provided that uses this imbalance information to balance the application via issue prioritization. We now describe these two distinct steps. For simplicity, we will describe the mechanism assuming a two-threaded in-order microprocessor. However, this algorithm can be extended to more than two threads.

4.1 Imbalance Identification

First, thread progress is maintained with the identification of the meeting point in the program where both threads reach. The natural location of this point is at the back edge of a parallelized loop. The meeting point is used by imbalance

hardware logic that detects, at runtime, the imbalance (see Figure 5). The imbalance hardware logic is responsible for monitoring the instruction that is currently executing to determine if the thread has reached the meeting point. If the thread has reached the meeting point, then it has completed another iteration of the loop. The output of the imbalance hardware logic consists of two things. Slow TID contains a pointer to the slower of the two threads, and Iteration Delta is a counter that contains the amount of iterations that the fast thread is ahead of the slow thread. In general, the logic notes the slow thread and keeps track of imbalance by adjusting the iteration delta appropriately. A simple state machine is proposed as the method to implement this imbalance hardware logic. The pseudocode of this logic is presented in Figure 5.

4.2 Issue Prioritization

The imbalance information is used by the issue prioritization hardware logic, which passes priority information to the issue stage of our in-order multi-threaded core (shown in Figure 4). The input to our mechanism is the iteration delta and slow TID information provided by the imbalance hardware logic described earlier. The output is a signal to prioritize the slow thread, allowing it to catch-up to the fast thread. The simple logic, implemented entirely in hardware, notes when there is a slow thread by comparing the iteration delta to a threshold. A threshold equal to 0 is found to provide the best results and is used in Figure 5. The pseudocode for our logic is given in Figure 5.

As an example of the entire idea, consider a benchmark with thread imbalance. A meeting point is determined either statically with the compiler or dynamically in hardware for the benchmark. The imbalance hardware logic would receive a disproportionate amount of meeting point IPs. It would identify and update the slow TID and accompanying iteration delta. The issue prioritization logic would observe this, informing the issue stage to give priority to the slow thread only until it catches up to the fast thread. If the slow thread catches up, iteration delta becomes 0, and the priority signal is not sent. The algorithm dynamically adapts based on execution.

4.3 A Separate Proposal with Thread Balancing for Energy Reduction

Instead of prioritizing issue bandwidth for performance improvement, a separate utilization of this thread imbalance logic is an energy reduction technique on an SMT microprocessor. This technique could be employed when issue prioritization is not balancing the threads appropriately, or if energy is a more important design metric.

Similar to the previous section, we utilize the hardware that identifies the slow thread. Then, during phases of inactivity for the slow threads such as a cache miss, instead of prioritizing issue bandwidth, we propose to power down or clock gate the entire processor for energy saving purposes.

This is demonstrated in Figure 5. The only difference is with the issue prioritization hardware logic. Instead of prioritizing issue bandwidth, clock gating will be employed upon a load miss. During this time, both threads will be turned off. As a result, energy will only be consumed during times when the slow

or critical thread is making forward progress. The overall number of energy-consuming cycles will be reduced. Therefore, overall energy consumed by the processor to execute these threads will be reduced. Furthermore, performance will not suffer significantly, since performance is dictated by the execution time of the slow thread.

Issue prioritization and energy reduction of thread balancing are not employed in the same hardware. We do not propose this as a single-joint design, but instead, either issue prioritization or clock gating can be used. Note, however, if thread balancing is working via issue prioritization, then there will be no energy savings possible for that particular application. In the next section, we describe which method should be used for each application that we study. A design that can switch between issue prioritization and clock gating based on application behavior is left for future work.

5. EXPERIMENTS

The simulation framework used in our study contains a full-system functional simulator and a performance simulator. SoftSDV [Uhlig et al. 1999] for Intel64/IA32 processors is our functional simulator, and it can simulate not only multithreaded primitives including locks and synchronization operations but also shared memory and events. Therefore, it is ideal to simulate our cooperative workload at the functional level. Redhat 3.0 EL is booted as the guest operating system in SoftSDV. In all of our simulations, only less than 1% of simulated instructions are from the operating system, and thus the impact of the operating system is minimal.

The functional simulator feeds Intel64/IA32 instructions into the performance simulator, which provides a cycle-accurate simulation. The performance simulator also incorporates a power model based on activity counters and energy per access, similar to Wattch [Brooks et al. 2000]. In our evaluation, the energy includes dynamic energy, idle energy, and leakage energy. The baseline assumes that every core is running at full speed and stops when it is completed. The cores will sleep after they reach a barrier. Once the core finishes, it consumes zero power.

Meeting point thread characterization, thread delaying, and thread balancing are implemented in our cycle-accurate performance simulator for a CMP or SMT system. Since thread delaying and thread balancing pursue different purposes and their effects are orthogonal, both techniques are evaluated independently. Thread delaying is evaluated for multicore systems where each core contains only one thread, while thread balancing is evaluated for a single SMT core (each core contains two threads). The simple in-order core is low power and is suitable for a many-core chip such as Sun's Niagara [Kongetira et al. 2005]. The detailed architectural parameters are shown in Table I.

5.1 Benchmarks

The RMS workloads from Intel are a set of emerging multithreaded applications for Tera-scale systems [Borkar 2005; Intel 2005]. The RMS workload includes highly compute-intensive and highly parallel applications including

Table I. The Architectural Parameters

Process Model	In-order Intel64/IA32
L1 Instruction Cache (private)	32KB, 4-way
L1 Data Cache (private)	32KB, 8-way
L2 Cache (unified and private)	512KB, 16-way
L3 Cache (unified and shared)	8MB, 16-way
Network Protocol	MESI

Table II. The RMS Benchmarks

Benchmark	Application
Gauss	Financial Analysis
PageRank (sparse)	Search Engine
PageRank (lz77)	Search Engine
Summarization	Text Data Mining
fimi	Data Mining
research	Bioinformatics
svm	Bioinformatics

computer vision, data mining on text and media, bioinformatics, and physical simulation.

From the RMS benchmark suite, we have chosen those that clearly show workload imbalance and one benchmark called Gauss, which is a relatively balanced workload. Gauss is chosen for testing the robustness of thread-delaying algorithm. These benchmarks are depicted in Table II. Gauss is a Gauss-Seidel iterative solver of a system of partial differential equations. The kernel of PageRank performs multiple matrix multiplications on a large and sparse matrix. The matrix can be stored in memory either in a native sparse or a compressed way. The compression is a simplified LZ77-based method. Summarization is a text data-mining workload, which finds and ranks documents in a Web search engine. fimi analyzes a set of data transactions, determining the rules related to the data. Both research and svm are used in bioinformatics to search in a database for both a homologous RNA and a disease gene pattern, respectively.

All of these workloads are already parallelized by using either pthreads or OpenMP to achieve maximal scalability. The benchmarks were developed by expert programmers and parallelized by hand. (i.e., OpenMP primitives are inserted by the programmer.) However, they still exhibit different degrees of workload imbalance and, therefore, inefficiency in the energy consumption.

The simulated section for each benchmark is chosen by first profiling its single-threaded counterpart and then selecting the hottest region, which normally is a parallel loop. For all of the benchmarks except fimi, the selected parallel regions represent almost 99% of total execution time. fimi has 28% coverage. In our simulation, each thread runs a fixed number of iterations, termed N and when the slowest thread has executed N iterations, the simulation is finished. The value of N varies depending on the benchmark. At least 100 million instructions are executed before a simulation is terminated.

Fig. 6. Performance results for thread delaying.

5.2 Performance Results for Thread Delaying

Figure 6 shows that thread delaying achieves significant energy reduction for selected RMS benchmarks under three different hardware configurations: two, four, and eight cores, ranging from 4% to 44% energy savings. In this experiment, each core executes one thread.

For most configurations, there is little performance loss, ranging from 1% to 2%. Moreover, there is even a case when thread delaying obtains speed-ups. Since all cores except the one containing the critical thread have their frequencies and voltages reduced, their cache misses are more spread out over time, allowing the critical thread to have more priority in the interconnection. This side effect of per-core DVFS accelerates the critical thread and thus reduces the total execution time.

5.3 Analysis of Thread-Delaying Performance

The first question that we must answer is where such energy savings come from. For example, PageRank (sparse) on eight cores achieves more than 40% energy savings. Figure 7 shows the runtime behavior of PageRank (sparse) before and after thread delaying. The x axis represents the number of iterations of the parallelized loop that each core executes. The y axis of Figures 7(a) and 7(b) represents the cumulative execution time of the loop iterations in milliseconds, whereas the y axis in Figure 7(c) represents the frequency of the core in GHz. We can see that there are large gaps between the critical thread (cpu0) and the rest of the threads. All noncritical threads except the one in cpu3 stay at the lowest frequency after iteration 6,600. For cpu3, it stays at the lowest frequency until iteration 12,200 and increases the frequency afterward, because the gap between cpu0 and cpu3 is getting smaller. It is obvious that the big energy

Fig. 7. PageRank (sparse) on eight cores. (a) Runtime behavior of the baseline. (b) Runtime behavior after applying thread delaying. (c) Corresponding frequency level.

Fig. 8. PageRank(lz77) on two cores. (a) Runtime behavior of the baseline. (b) Runtime behavior after applying thread delaying. (c) Corresponding frequency level.

savings come from the large frequency decreases on noncritical threads. Similar observations are also obtained for the PageRank (sparse)’s 4p configuration and Summarization’s 4p and 8p configurations.

The effectiveness of thread delaying depends on whether the algorithm can adapt quickly at runtime; in other words, the algorithm chooses frequencies in a way that reflects the runtime behavior of the application. To demonstrate this, we use the example in Figure 8. In Figure 8 between iteration 10 and iteration 40, the time gap becomes smaller and smaller and our algorithm increments the frequency of the noncritical thread slowly. By doing that, the noncritical thread can avoid staying at a low frequency level for too long and

Fig. 9. Gauss on two cores (a) Runtime behavior of the baseline (b) Runtime behavior after applying thread delaying (c) corresponding frequency level.

becoming a false-critical thread. If the noncritical thread became a false-critical thread, there would be performance penalty at the end. At iteration 65, there is a cache miss with long latency, which results in a time difference between two threads again. Our algorithm immediately observes this change and starts to decrement the frequency level of the noncritical thread. The frequency of `cpu1` (the critical thread) is slightly scaled down from 4GHz to 3.75GHz (see iterations between iterations 60 and 65). However, our mechanism can quickly correct the mistake once there is a time gap between these two threads. After iteration 65, the frequency of critical thread is back to the maximum.

We have demonstrated that large energy savings can be obtained from imbalanced workloads. Moreover, our thread-delaying algorithm can also save energy as much as possible for relatively balanced workloads. For example, Figure 9 shows the runtime behavior before and after thread delaying for Gauss. Gauss is a balanced workload and it is hard to distinguish which threads are critical or noncritical (four lines in Figure 9(a) are totally overlapped). However, from Figure 9(c), we can see that `cpu0` is the critical thread. By reducing the frequency of noncritical thread to next frequency level, we still can achieve 6% energy savings without any performance penalty.

From previous observations, we can see that our thread delaying is robust and effective. It can maximize the energy savings with negligible performance loss.

5.4 Performance Results for Thread Balancing on an SMT

We now shift our focus to the results of thread balancing on a simultaneous multithreaded microprocessor. We first begin by identifying the level of thread imbalance for our RMS benchmarks. Figure 10 shows the level of imbalance for four RMS benchmarks. These results assume a two-threaded core running a two-threaded application. To calculate imbalance, we first count the number of iterations that are executed by the slow thread after the fast thread has finished execution. Second, we divide that count by the number of total iterations in the

Fig. 10. Thread imbalance.

loop to obtain imbalance. The level of imbalance varies for these workloads; however, all have imbalances greater than 7%, ranging up to 50%. We have studied the causes to thread imbalance and determined that the key cause is the imbalance in cache behavior between the threads. This cause is shown in Figure 11, where the percentage of issue stalls from load instructions is shown. Note that PageRank(sparse) has a high percentage of load miss stalls. The slow thread often has many more cache misses than the fast thread. Different code flow behavior also contributes significantly to this thread imbalance.

Results of the performance benefit of our mechanism over the baseline for four RMS workloads are shown in Figure 12. Performance benefit ranges from 0.8% to 20%, and the harmonic mean average is 7%. Generally speaking, the performance benefit correlates with imbalance levels given in Figure 10. For fimi as an example, there is a large level of thread imbalance and a corresponding amount of performance improvement by administering issue priority to the slower thread.

5.5 Analysis for Thread Balancing on an SMT

We begin our analysis by determining the efficacy of this algorithm. We first present in Table III, the correction percentage, or the level of balance when applying the algorithm. Correction of imbalance is measured by determining the number of iterations that the slow thread has completed when the fast thread has finished, and comparing that to the baseline level of imbalance. For example, assume, in the baseline algorithm, the slow thread completes 5 iterations and the fast thread 10. Then, with the algorithm, the slow thread completes 8 iterations and the fast thread 10. The correction percentage would be 60%, since three fifths of the imbalance is corrected. As shown in the table, fimi and svmclass have 100% imbalance correction with this algorithm and

Fig. 11. Thread imbalance.

Fig. 12. Performance improvement.

are operating in a fully balanced situation. We can do no more to correct these benchmarks.

It can be seen that only 49% of imbalance is corrected for rsearch. Figure 13 shows the percentage of cycles that the slow thread is not consuming full issue bandwidth when it has ready instructions. Put another way, it identifies the amount of cycles that the slow thread was not given all issue slots when it could have been. A number of 0% means that the slow thread, when it had ready instructions was given full-issue bandwidth all of the time. For example, for fimi, it can be seen that without the algorithm 25% of cycles, the slow thread had ready instructions but had to split issue bandwidth with the fast thread. However, with the balancing algorithm, this number is reduced to less than 5% leading to a performance improvement. For rsearch, this number is reduced to 0%. However, although our algorithm has reached its maximum potential, it

Table III. Level of Balance with Algorithm

	fimi	rsearch	svmclass	PageRank
correction (%)	100	49	100	1

Fig. 13. Issue stall improvement.

has not balanced this application. Previously, the application stalls at the issue stage, splitting issue bandwidth with the fast thread. Now, it is stalling at the issue stage for other reasons, including instructions not available, the store buffer is full, load misses or dependency stalls. To get to an ideal situation for rsearch, we would need to apply our priority algorithm to other facets of the microprocessor.

Also shown in Figure 13 is that for PageRank, there is also little opportunity for balancing. Less than 2% of all cycles does the slow thread have ready instructions to consume full-issue bandwidth. Again, this is due to repeated cache misses for this benchmark. We will discuss PageRank in more detail in the next section.

By changing the issue scheduling order, we are indirectly impacting data cache access order. Without our balancing algorithm, the fast thread accesses the cache repeatedly while the slow thread is stalled. Now, with the balancing algorithm, the slow thread accesses the cache more consistently, potentially increasing its hit rate. However, the overall hit rate of both threads may be reduced because the threads may have previously shared data synchronously. We now present the data cache effects of our algorithm. Shown in Figure 14 is the affect on load misses the algorithm causes. Specifically, the ratio of total cache misses without the algorithm to cache misses with the algorithm for each benchmark is plotted. For fimi, we find that we have approximately 5% more misses with our algorithm. Changing the order of cache accesses can have an overall negative affect on cache behavior. This, however, did not have an

Fig. 14. Data cache impact.

overall negative affect on fimi's performance. Conversely, for svmclass, cache misses are reduced by 8%, leading to better overall performance. However, cache misses are not affected for both rsearch and PageRank.

5.6 Energy Savings with Thread Balancing on an SMT

Thread imbalance could not be corrected for PageRank. The opportunity to correct the imbalance is just not there (see Figures 11 and 13). Load misses are abundant for the slow thread, and we do not have the opportunity to steal instruction bandwidth from the fast thread. For this benchmark, where thread imbalance does not improve with our balancing technique, we instead employ our thread-balancing energy-saving technique, as described in Section 4.3 (note: This is different than our thread-delaying proposal). We measure that 33% of all cycles can be clock gated during times of load miss for the slow thread for PageRank. Energy is only consumed when both threads are making forward progress. Overall energy can be saved because the core is being fully utilized when powered on, and idle power can, therefore, be reduced.

As described previously, we have implemented a power and energy simulation model in our simulator. During the thread-balancing clock-gated time, we assume that neither thread is executing. We measured energy consumption without our algorithm compared to the energy consumption with our thread-balancing energy-saving algorithm. With our model, by clock-gating the core during all stalled time for the slow thread, we were able to save 23.8% total energy when compared to the baseline scheme that does not implement clock-gating.

Performance can degrade in an SMT core when clock gating is employed because the fast thread could have continued executing and left more available resources for the slow thread once the slow thread is no longer stalled. We

Table IV. Clock Gating for Thread Balancing

	Cycles clock-gated	Energy savings	Performance degradation
PageRank	33%	23.8%	3%

measured performance without our algorithm compared to the performance with our thread-balancing energy-saving algorithm. We found that performance with thread-balancing, clock gating only suffers 3% degradation. When thread balancing by issue prioritization is not effective, then energy consumption can be reduced by exploiting thread imbalance on a simultaneous multi-threaded microprocessor.

6. RELATED AND FUTURE WORK

There are previous works related to thread delaying. Liu et al. [2005] proposed an algorithm, which tracks the time spent by the faster cores waiting for the slower cores at the end of a parallel loop and predicts the DVFS level of each core for the next execution of the same parallel loop. The main difference between our thread-delaying approach and the one proposed by Liu et al. [2005] is that our approach runs at a finer grain, adapting to runtime behavior inside the execution of the parallel loop. Following from this key difference, our mechanism can handle the cases that their mechanism cannot handle because we do not require multiple instances of a loop. It is not because the loop is insignificant, but because the parallelized section (not loop) is only executed once in the whole application. For example, in Summarization, the parallelized section looks as follows:

```
#pragma omp parallel
{
    while (n < Niterations) {
        # pragma omp barrier
        {}
        // a lot of codes
    }
}
```

This parallelized section is only executed once, but it is the hottest one (99% of total execution). Each thread will execute loop iterations many times. Our approach can handle this case, but their mechanism [Liu et al. 2005] cannot. Additionally, our meeting-point algorithm provides an opportunity to apply thread balancing on an SMT core. The scheme in Liu et al. [2005] does not provide this opportunity.

The second work related to our work is called the thrifty barrier [Li et al. 2004]. The thrifty barrier uses the idleness at the barrier to move the faster cores to a low power mode. It has been shown that the DVFS approach outperforms the thrifty barrier approach [Liu et al. 2005]. Furthermore, our baseline can be considered as an aggressive version of thrifty barrier, since when a thread arrives to a barrier, it consumes zero power.

Thread delaying is motivated by the workload imbalance among parallel threads. This type of performance asymmetry due to workload imbalance is different from the performance asymmetry discussed in the literature [Balakrishnan et al. 2005; Kumar et al. 2003; Kumar et al. 2004]. They created a performance-asymmetric multicore system, including high-performance complex core and low-performance simple cores, in such a way that the complex cores provide good serial performance and simple cores provide high throughput. However, in our case, the asymmetry comes from the workload imbalance among parallel threads from the same parallel region. As the workload imbalance is mainly due to cache misses from our experiments, many simple cores are enough for highly parallel and computationally intensive applications such as RMS, and complex and powerful cores do not help to speed up the performance or save energy in this case. Therefore, we are addressing the problem different from Balakrishnan et al. [2005] and Kumar et al. [2003, 2004].

The DVFS algorithm can also be implemented at the operating system level. Lorch and Smith [2001] proposed a scheduling algorithm, which schedules a task in such a way that the frequency/voltage of a CPU is scaled down to save energy and meet the deadline of the task. There are three big differences between this work and our work. First, the deadline of a task is not known, and it is decided manually. Our meeting point thread characterization can select the critical thread dynamically, and the critical thread actually determines the deadline of the whole parallel execution. Second, the workloads they use are mostly interactive benchmarks, such as word processing and spread sheet, which are very different from our highly parallel RMS applications. Third, their algorithm is an OS-scheduling algorithm for only one CPU, whereas our algorithms are lightweight enough to be implemented in a hardware targeting many-core system.

Another recent work that is similar in spirit is CAP [Tuck et al. 2007]. This article introduces a novel task-criticality model for Speculative Multithreading and uses it to make scheduling decisions. Specifically, they predict criticality of threads with a complex dynamic predictor and use this information to schedule noncritical threads on lower power cores. There are three main differences with this work and our scheme. First, our scheme is not predicting but very efficiently identifying the threads that are falling behind. Second, our model is a very simple in-order microprocessor, very different from their Speculative Multithreaded machine. Finally, we introduce the thread-balancing technique, which is significantly different than thread delaying or CAP. The purpose of our meeting point identification is to identify loops that are parallelized into threads. Although not shown in this article, it can be applied to nonopenMP loops in software. Our identification can be described as a subset of the CAP approach, where all threads are identified as critical or not. If a broader coverage of all thread types is desired, then our thread-balancing technique could be combined with the CAP approach.

Our work on thread balancing is unique. We are not aware of any research that is similar to this new mechanism. There has been an abundance of research focusing on thread prioritization [Cazorla et al. 2004; El-Moursy and Albonesi 2003; Homayoun et al. 2005; Jain et al. 2002; Robatmili et al. 2004; Tullsen

et al. 1996]. However, the focus is on prioritizing threads that are ready to execute, that is, the fast threads. Prior art does not consider threads that are imbalanced from the same application. Our goal is the opposite, trying to give priority to the slower threads, noting that the slow threads dictate the overall performance of the application.

Our scheme could be limited by the lack of ability of communication between the threads, similar to scalability problems that parallel data-sharing threads could encounter. For a futuristic processor that contains hundreds of cores, our scheme can still be applied by grouping cores together, assigning energy on a higher-granularity scale. An exploration of this is outside the scope of this study and is left for future work.

7. CONCLUSION

After introducing a mechanism to identify critical threads, we provide an analysis of thread-delaying and thread-balancing techniques. Our experiments with several recognition, mining, and synthesis (RMS) workloads show that thread delaying on a CMP system can greatly reduce energy (from 4 to 44%) with negligible performance penalty. Our thread-balancing mechanism can improve performance for various RMS workloads, ranging up to 20%.

In this work, we provide a detailed breakdown of performance results for thread balancing, identifying performance benefits and limitations. We illustrate causes of stalls, levels of imbalance, and reasons for performance limitations. For benchmarks where a performance benefit is not possible, we introduce a novel thread-balancing mechanism on an SMT core that can reduce energy consumption. It is based on thread-balancing logic that turns off both threads when the slow thread is stalled. With this novel approach, we can save up to 23% of energy consumption with a slight degradation in performance.

Our future work includes looking at other areas of the microprocessor in order to shift priority from the fast threads to slow threads. For example, prioritizing the data cache is one such approach. In the same spirit, we plan to take a deeper look at energy-saving techniques, including additional reasons to clock gate the microprocessor. Our invention to correct thread imbalance is simple, requires a small amount of the hardware, and can be used as a general framework for these future energy-saving techniques.

REFERENCES

- BALAKRISHNAN S., RAJWAR, R., UPTON, M., AND LAI, K. 2005. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. IEEE, Los Alamitos, CA, 506–517.
- OPENMP ARCHITECTURE REVIEW BOARD. 2005. OpenMP application program interface.
- BORKAR, S. Y. 2005. Platform 2015: Intel processor and platform evolution for the next decade. Intel White Paper.
- BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. *SIGARCH Comput. Archit. News*, 28.
- BRUD, T. D. AND BRODERSEN, R. W. 1995. Energy efficient cmos microprocessor design. In *Proceedings of the 28th Hawaii International Conference on System Sciences*. IEEE, Los Alamitos, CA.

- CAZORLA, F. J., RAMIREZ, A., VALERO, M., AND FERNANDEZ, E. 2004. Dynamically controlled resource allocation in SMT Processors. In *Proceedings of the International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA.
- CAI, Q., GONZALEZ, J., RAKVIC, R., MAGKLIS, G., CHAPARRO, P., AND GONZALEZ, A. 2008. Meeting points: Using thread criticality to adapt multicore Hardware to parallel regions. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, Los Alamitos, CA.
- CHANEY, T. J. AND MOLNAR, C. E. 1973. Anomalous behavior of synchronizer and arbiter circuits. *Trans. Comput.* 22, 4.
- CHAPARRO, P., GONZALEZ, J., MAGKLIS, G., CAI, Q., AND GONZALEZ, A. 2007. Understanding the thermal implications of multicore architectures. *Trans. Paralle. Distrib. Syst.* 18, 8.
- CHELCEA, T. AND NOWICK, S. M. 2001. Robust interfaces for mixed-timing systems with application to latency-insensitive protocols. In *Proceedings of the 38th Design Automation Conference*. ACM, New York.
- EL-MOURS, A. AND ALBONSESI, D. H. 2003. Front-end policies for improved issue efficiency in SMT processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA.
- FISCHER, S. 2007. Technical overview of the 45nm next generation Intel core microarchitecture (Penryn). EMEA Academic Forum.
- FISCHER, T., DESAI, J., DOYLE, B., NAFFZIGER, A., AND PATELLA, B. 2006. A 90-nm variable frequency clock system for a power-managed Itanium architecture processor. *J. Solid-State Circuits*, 41.
- GOCHMAN, S., RONEN, R., ANATI, I., BERKOVITS, A., KURTS, T., NAVEH, A., SAEED, A., SPERBER, Z., AND VALENTINE, R. 2003. The Intel Pentium m processor: Microarchitecture and performance. *Intel Tech. J.* 7, 2.
- HAZUCHA, P., KARNIK, T., BLOECHER, B. A., PARSONS, C., FINAN, D., AND BORKAR, S. 2005. Area-efficient linear regulator with ultra-fast load regulation. *J. Solid-State Circuits*, 40.
- HOMAYOUN, H., LI, K. F., AND RAFATIRAD, S. 2005. Thread scheduling based on low-quality instruction prediction for simultaneous multithreaded processors. In *Proceedings of the 3rd International NEWCAS Conference*. IEEE, Los Alamitos, CA.
- HU, C. 1994. Low-voltage cmos device scaling. In *Proceedings of International Solid-State Circuits Conference*. IEEE, Los Alamitos, CA.
- INTEL CORPORATION. 2005. Computer intensive, highly parallel application and uses. *Intel Tech. J.*, 9, 2.
- INTEL CORPORATION. 2006. Intel's tera-scale research prepares for tens, hundreds of cores.
- IYER, A. AND MARCULESCU, D. 2002. Power and performance evaluation of globally asynchronous locally synchronous processors. *SIGARCH Comput. Archit. News* 30.
- JAIN, R., HUGHES, C., AND ADVE, S. 2002. Soft real-time scheduling on simultaneous multithreaded processors. In *Proceedings of the 23rd International Real-Time Systems Symposium*. IEEE, Los Alamitos, CA.
- KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. 2005. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25.
- KUMAR, R., FARKAS, K., JOUPPI, N., RANGANATHAN, P., AND TULLSEN, D. 2003. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 81.
- KUMAR, R., TULLSEN, D., RANGANATHAN, P., JOUPPI, N., AND FARKAS, K. 2004. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*. IEEE, Los Alamitos, CA.
- LI, J., MARTINEZ, J., AND HUANG, M. 2004. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA.
- LIU, C., SIVASUBRAMANIAM, A., KANDEMIR, M., AND IRWIN, M. 2005. Exploiting barriers to optimize power consumption of cmps. In *Proceedings of the Parallel and Distributed Processing Symposium*. IEEE, Los Alamitos, CA.
- LORCH, J. AND SMITH, A. 2001. Improving dynamic voltage scaling algorithms with pace. In *Proceedings of the ACM SIGMETRICS Conference*. ACM, New York.

- MAGKLIS, G., CHAPARRO, P., GONZALEZ, J., AND GONZALEZ, A. 2006. Independent front-end and back-end dynamic voltage scaling for a gals microarchitecture. In *Proceedings of the International Symposium on Low-Power Electronics and Design*. ACM, New York.
- MAGKLIS, G., GONZALEZ, J., AND GONZALEZ, A. 2004. Frontend frequency-voltage adaptation for optimal energy-delay2. In *Proceedings of the International Conference on Computer Design*. IEEE, Los Alamitos, CA.
- MARCUELLO, P., GONZALEZ, A., AND TUBELLA, J. 1998. Speculative multithreaded processors. In *Proceedings of the International Conference on Super-Computing*. ACM, New York.
- OLSSON, T., NILSSON, P., MEINKE, T., HEMAM, A., AND TORKELSON, M. 2000. A digitally controlled low-power clock multiplier for globally asynchronous locally synchronous designs. In *Proceedings of the International Symposium on Computer Architecture*. IEEE, Los Alamitos, CA.
- OLUKOTUN, K., NAYFEH, B. A., HAMMOND, L., WILSON, K., AND CHANG, K. 1996. The case for a single-chip multiprocessor. *SIGOPS Oper. Syst. Rev.* 30.
- ROBATMILI, B., YAZDANI, N., SARDAHTI, S., AND NOURANI, M. 2004. Thread-sensitive instruction issue for smt processors. *Comput. Archit. Lett.* 3.
- SEMERARO, G., ALBONESI, D., MAGKLIS, G., SCOTT, M., DROSHO, S., AND DWARKADAS, S. 2004. Hiding synchronization delays in a gals processor microarchitecture. In *Proceedings of the 10th International Symposium on Asynchronous Circuits and Systems*. IEEE, Los Alamitos, CA.
- TUCK, J., LIU, W., AND TORRELLAS, J. 2007. CAP: Criticality analysis for power-efficient speculative multi-threading. In *Proceedings of the International Conference on Computer Design*. IEEE, Los Alamitos, CA.
- TULLSEN, D., EGGERS, S., EMER, J., LEVY, H., LO, J., AND STAMM, R. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. *SIGARCH Comput. Archit. News* 24.
- UHLIG, R., FISHTEN, R., GERSHON, O., HIRSH, I., AND WANG, H. 1999. Softsdv: A pre-silicon software development environment for the ia-64 architecture. *Intel Tech. J.* 3, 4.
- WU, Q., JUANG, P., MARTONOSI, M., AND CLARK, D. 2005. Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA.
- ZHU, W., CUVILLO, J., AND GAO, G. 2006. Performance characteristics of OpenMP language constructs on a many-core-on-a-chip architecture. In *Proceedings of the 2nd International Workshop on OpenMP*.

Received January 2009; revised October 2010; accepted March 2010