Meeting Points: Using Thread Criticality to Adapt Multicore Hardware to Parallel Regions

Qiong Cai[1], José González[1], Ryan Rakvic[2], Grigorios Magklis[1], Pedro Chaparro[1] and Antonio González[1]
[1] Intel Barcelona Research Center, Intel Labs-UPC, Barcelona, Spain
{qiongx.cai, pepe.gonzalez, grigorios.magklis, pedro.chaparro.monferrer, antonio.gonzalez}@intel.com
[2] United States Naval Academy, Annapolis, Maryland, USA
rakvic@usna.edu

ABSTRACT

We present a novel mechanism, called meeting point thread characterization, to dynamically detect critical threads in a parallel region. We define the critical thread the one with the longest completion time in the parallel region. Knowing the criticality of each thread has many potential applications. In this work, we propose two applications: thread delaying for multi-core systems and thread balancing for simultaneous multi-threaded (SMT) cores. Thread delaying saves energy consumptions by running the core containing the critical thread at maximum frequency while scaling down the frequency and voltage of the cores containing non-critical threads. Thread balancing improves overall performance by giving higher priority to the critical thread in the issue queue of an SMT core. Our experiments on a detailed microprocessor simulator with the Recognition, Mining, and Synthesis applications from Intel research laboratory reveal that thread delaying can achieve energy savings up to more than 40% with negligible performance loss. Thread balancing can improve performance from 1% to 20%.

Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: Multiple Data Stream Architectures – *parallel processors.*

General Terms

Performance, Design

Keywords

Meeting point thread characterization, Critical threads, Thread delaying, Thread balancing, Multi-threaded Application, Microarchitecture, Low-power, Energy-aware

1. INTRODUCTION

In recent years, chip multiprocessors (CMPs) [31] have become increasingly important and common in the computer industry [3][11][21]. The design of CMP processors takes advantage of thread-level parallelism (TLP) to address the problem of limited instruction-level parallelism (ILP) in serial applications. Moreover, it is believed that future applications will be compute-intensive and highly parallel [10]. For example, the Intel Tera-scale research [11] aims at integrating tens or hundreds of cores in a future microprocessor, to run highly parallel workloads such as Recognition, Mining, and Synthesis (RMS) applications [10].

High energy consumption will be one of the major hurdles 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 [2], a parallel loop usually has a barrier at the joint 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. In order to reduce energy, one possible solution is to put fast threads to sleep as soon as they arrive to the barrier and then shut down the core. This is a feasible approach if the waiting time is long enough so that the energy saved in sleep mode pays off the energy/performance wasted by putting the cores to sleep and waking them up. We propose an alternative solution in this paper. If a thread is known beforehand to reach the synchronization point early, the voltage and frequency of the core running that thread could be reduced dynamically without compromising performance.

As we will demonstrate later in this work, dynamic voltage and frequency scaling (DVFS) achieves greater energy reduction compared to putting a core to sleep due to the cubic relationship of power to voltage/frequency. One of the main challenges in this approach is the detection of the critical and non-critical threads. We use the ***slack*** as a proxy to know the criticality level of a parallel thread. We define slack for a parallel thread as the amount of time a thread can be delayed with no impact on final performance. The critical thread is the one with zero slack, which means that if it gets delayed, the overall execution time is increased. Non-critical threads are those that could be delayed with no impact on performance. The level of criticality is determined by the amount of slack that each thread has. Detecting such critical threads and the level of criticality is challenging, since one could not know a priori whether a thread is going to be the last one to reach the barrier.

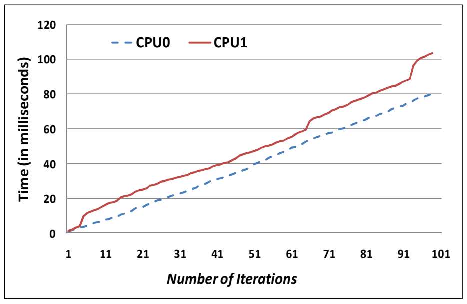
In this paper, we present a novel mechanism called ***meeting point thread characterization*** that identifies the critical thread of a single multi-threaded application as well as the amount of the slacks of non-critical threads. To do that, each thread has a counter to accumulate the number of iterations executed for the parallel loop. At specific intervals of time, all threads broadcast

```
#pragma omp parallel for
for (int i = 0; i < nb ; i++ )
{
    Msizeu partstart = partition()[i];
    Msizeu partend = partition()[i+1];

    diag(i).multply_transposed(...);
    offdiag.multply(...);

    for (int i = partstart; i < partend; i++)
    {
          res[i] *=c;
          normres += std::abs(res[i]);
    }
}
```

(a)



(b)

```
#pragma omp parallel for
for (int i = 0; i < nb ; i++ )
{
    Msizeu partstart = partition()[i];
    Msizeu partend = partition()[i+1];

    diag(i).multply_transposed(... );
    offdiag.multply(...);

    for (int i = partstart; i < partend; i++)
    {
          res[i] *=c;
          normres += std::abs(res[i]);
    }

    asm( "mp_inst" );
}
```

(c)

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

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. In this work, we focus on two of such optimizations that dynamically adapt the hardware resources to the application behavior: ***thread delaying*** and ***thread balancing***.

The goal of thread delaying is to reduce overall energy consumption by dynamically scaling down the voltage and the frequency of the cores executing non-critical 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. These architectures usually implement fair policies regarding shared resources. For instance, it is common to share the issue slots in such a way that, if both threads have ready instructions, both are allowed to issue the same number of them. 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 [5][12][17][20][32][34]. 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.

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. For example, for PageRank, which represents an important category of emerging applications such as Google's web search engine, thread delaying can achieve more than 40% energy savings without any performance loss on an eight-core system. It is important to note that the baseline is very aggressive, since we assume that, once a thread arrives to a barrier, its core is set instantaneously to deep sleep state [13], consuming zero power. Moreover, the experiments on an SMT in-order core show that our thread balancing mechanism can improve performance for various RMS workloads, from 1% to 20%.

In the rest of paper, we first describe the mechanism of identifying critical threads in parallel applications in Section 2. The thread delaying and thread balancing techniques are explained in detail in Sections 3 and 4, respectively. Section 5 describes our simulation framework and shows the performance results of thread delaying and thread balancing. We also discuss the related work in Section 6. The paper concludes 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 (each core contains one thread). The x-axis in Figure 1 (b) 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 path 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 check points ***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 [2], 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 [37]. The decision whether to use static or dynamic scheduling in a parallel is out of the scope of this paper.

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

- **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 last statement of the outermost loop (or the parallelized loop) satisfies our criteria.

  The insertion of a meeting point can be done by the hardware, the compiler or the programmer. A hardware-only approach, although it 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 [29]).

- **Identification of critical threads**: Every time a core decodes an instruction encoding a meeting point, a thread-private counter (located in the processor frontend) 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 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.

- **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.

In the next sections, we will describe two specific applications of the meeting point technique. More importantly, we will show how criticality information can be effectively used for different purposes, either for energy savings or for performance speedups.

3. THREAD DELAYING

As we 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 non-critical threads finish their jobs earlier than their critical counterparts do. Since there is a barrier at the join point of a parallelized loop, non-critical threads will have to wait (doing nothing) for the critical thread to finish its work, before they can proceed. In modern systems, the CPUs, of the non-critical threads, can be put into deep sleep mode, which consumes almost zero energy [13]. 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 non-critical 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 non-critical thread can finish its work in only ***0.7T*** time units. If the non-critical thread 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 non-critical thread is given by the following formula:

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

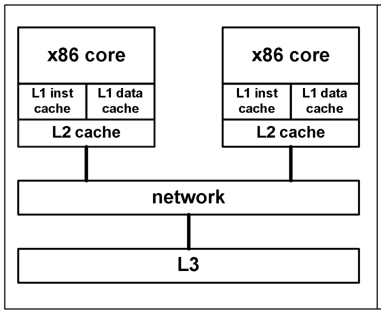
Alternatively, the core running the non-critical thread can have its frequency scaled-down to $\mathbf{0.7f_{max}}$ (and the voltage to $\mathbf{0.875V_{dd}}$, see Figure 2 (b)) and it would meet the barrier on time anyway. In this case, the total consumption for the non-critical thread is as follows:

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

From the above deductions, we can clearly see the advantage of doing DVFS on non-critical threads. There are two main challenges by applying DVFS in this scenario. First, we need a way to identify non-critical and critical threads at runtime. Second, we need to select appropriate frequency and voltage levels for non-critical 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

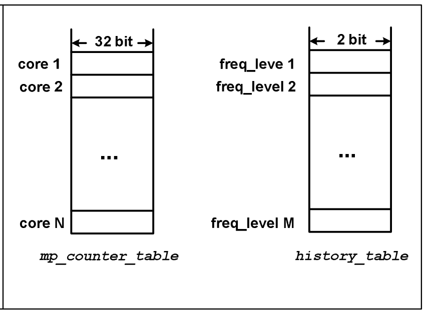
Figure 2(a) shows the baseline of our CMP microarchitecture. Our CMP processor consists of many Intel64/IA32 cores, and each core is a single-threaded in-order core (with bandwidth of 2 instructions per cycle) due to power and temperature constraints. Every core contains a private first-level instruction 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 (further parameters of the microarchitecture will be detailed in the Section 5).



(a)

Level	mV	GHz
0	700	0.90
1	725	1.20
2	750	1.45
3	775	1.75
4	800	2.00
5	825	2.25
6	850	2.55
7	875	2.80
8	900	3.05
9	925	3.30
10	950	3.50
11	975	3.75
12	1000	4.00

(b)



(c)

Figure 2 (a) Our CMP microarchitecture (b) Voltage-Frequency Table (c) Two tables are required to implement thread delaying

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. This allows firstly to run each domain at a different frequency and secondly 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 meta-stability [7]. We use the mixed-clock FIFO design of Chelcea and Nowick to communicate values safely between domains [9].

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 [19][28][33][36] and some commercial designs [15]. 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 [14][30]. 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 (a) only a few frequency levels are available, and (b) 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 (recall that the user inserts meeting points, which are represented by special instructions):

- **`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 two-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 execution 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 no 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 (which 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.

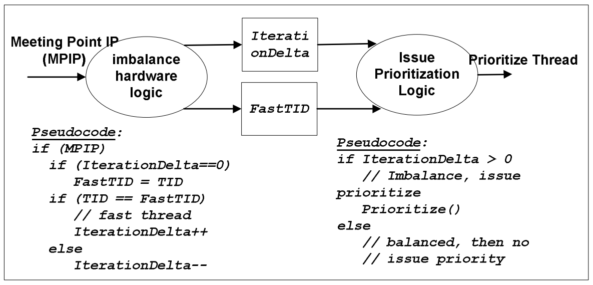
Every 10 executions of the meeting point instruction, the processor frontend stops fetching instructions and inserts a microcode (stored in a local ROM) to execute the thread delaying control algorithm. This microcode has dozens of instructions and its overhead has no impact on final performance. That microcode has as input both the **`MP-COUNTER-TABLE`** and the **`HISTORY-TABLE`** and its output is the frequency f_i 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 non-critical thread i, respectively. After f_{temp} is obtained, f_i is calculated by finding the minimum frequency supported in the



```
Pseudocode:
if (MPIP)
  if (IterationDelta==0)
     FastTID = TID
  if (TID == FastTID)
     // fast thread
     IterationDelta++
  else
     IterationDelta--
```

```
Pseudocode:
if IterationDelta > 0
   // Imbalance, issue
prioritize
   Prioritize()
else
   // balanced, then no
   // issue priority
```

Figure 3 Implementation of Thread Balancing Logic

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 [8][14][27][30].

Once the frequency level for f_i is obtained, the **HISTORY-TABLE** is updated properly (each entry contains a two 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 non-critical 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 non-critical threads. In this section, we focus on speeding up a parallel application running more than one thread on a single 2-way SMT core by accelerating the critical thread.

In an SMT core, the issue bandwidth is limited and shared among threads. There are a lot of issue policies in the literature [12][17][20][32][34], most of which assume that threads come from different applications (multi-programmed workloads). The baseline issue logic we have implemented works as follow (our microarchitecture is two-way in-order SMT with an issue bandwidth of two instructions per cycle): if both threads have ready instructions, each one of them is allowed to issue 1 instruction. If one thread has ready instructions and the other does not, the one with ready instruction can issue up to two per cycle. This algorithm 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.

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

Table 1 The architectural parameters

Figure 3 shows the implementation of thread balancing in hardware. Note that the scheme we propose regarding thread balancing works at the core level. Given two threads in a SMT core, it determines which is the critical one and gives more priority to this thread in the issue logic.

The hardware required to support thread balancing is simpler than thread delaying. Two hardware counters located in the processor frontend suffice to detect the critical thread between two threads running in the same core.

Every time the processor decodes a meeting point (inserted by the user, as aforementioned) the counter associated with that thread is increased. Every 10 executions of the meeting point instruction, both counters are compared. If the difference is greater than a given delta, the thread with the lowest counter value is designated as the critical thread, and that information is forwarded to the issue logic in the core.

The issue logic implements the fair policy detailed at the beginning of the section. However, if the frontend informs the issue logic that a given thread is critical, the issue policy is changed. If the critical thread has two ready instructions, it is allowed to issue both instructions regardless of the number of ready instructions the non-critical thread has. If it does not have two ready instructions, the base policy is applied.

Our thread balancing mechanism has potential speedup benefits when each thread follows different control paths from the same parallelized region. The critical thread (or the slowest thread) has more work to perform before reaching the meeting point than the non-critical thread has. In other words, when the critical thread is not slowed down by cache misses, thread balancing can speed up the whole application on an SMT core.

5. EXPERIMENTS

The simulation framework used in our study contains a full-system functional simulator and a performance simulator. SoftSDV [35] 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.

Benchmark	Application
Gauss	Financial Analysis
PageRank (sparse)	Search Engine
PageRank (lz77)	Search Engine
Summarization	Text Data Mining
FIMI	Data Mining
Rsearch	Bioinformatics
SVM	Bioinformatics

Table 2 The RMS Benchmarks

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 [4]. 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. Once the core stops, it consumes zero power.

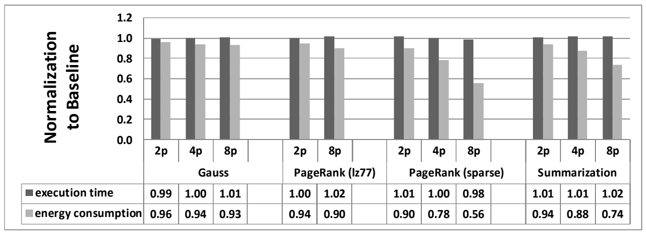
Meeting point thread characterization, thread delaying and thread balancing are implemented in our cycle-accurate performance simulator for a CMP or SMT system. Since thread delaying and thread balancing pursue different purposes and their effects are orthogonal, both techniques are evaluated independently. Thread delaying is evaluated for multi-core 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 [21]. The detailed architectural parameters are shown in Table 1.

5.1 Benchmarks

The Recognition, Mining, and Synthesis (RMS) workloads from Intel are a set of emerging multi-threaded applications for Tera-scale systems [3][10]. The RMS workload includes highly compute-intensive and highly parallel applications including computer vision, data mining on text and media, bio-informatics and physical simulation.

From the RMS benchmark suite, we have chosen those that clearly show workload imbalance and one benchmark called Gauss, which is relatively balanced workload. Gauss is chosen for testing the robustness of thread delaying algorithm. These benchmarks are depicted in Table 2. 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 Rsearch 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.



	Gauss 2p	Gauss 4p	Gauss 8p	PageRank (lz77) 2p	PageRank (lz77) 8p	PageRank (sparse) 2p	PageRank (sparse) 4p	PageRank (sparse) 8p	Summarization 2p	Summarization 4p	Summarization 8p
■ execution time	0.99	1.00	1.01	1.00	1.02	1.01	1.00	0.98	1.01	1.01	1.02
■ energy consumption	0.96	0.94	0.93	0.94	0.90	0.90	0.78	0.56	0.94	0.88	0.74

Figure 4 Performance Results for Thread Delaying

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 (say 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 (sum of instructions from all threads) are executed before a simulation is terminated.

5.2 Performance Results for Thread Delaying

Figure 4 shows that thread delaying achieves significant energy reduction for selected RMS benchmarks under three different hardware configurations: two, four (We had difficulty to run the simulation for four cores PageRank (lz77). So this configuration is excluded from our experiment.) 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 speedups. 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 5 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 Figure 5(a) and (b) represents the cumulative execution time of the loop iterations in milliseconds, whereas the y-axis in Figure 5(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 non-critical threads except the one in cpu3 stay at the lowest frequency after iteration 6600. For cpu3, it stays at the lowest frequency until iteration 12200 and increases the frequency afterwards, because the gap between cpu0 and cpu3 is getting smaller. It is obvious that the big energy savings come from the large frequency decreases on non-critical threads. Similar

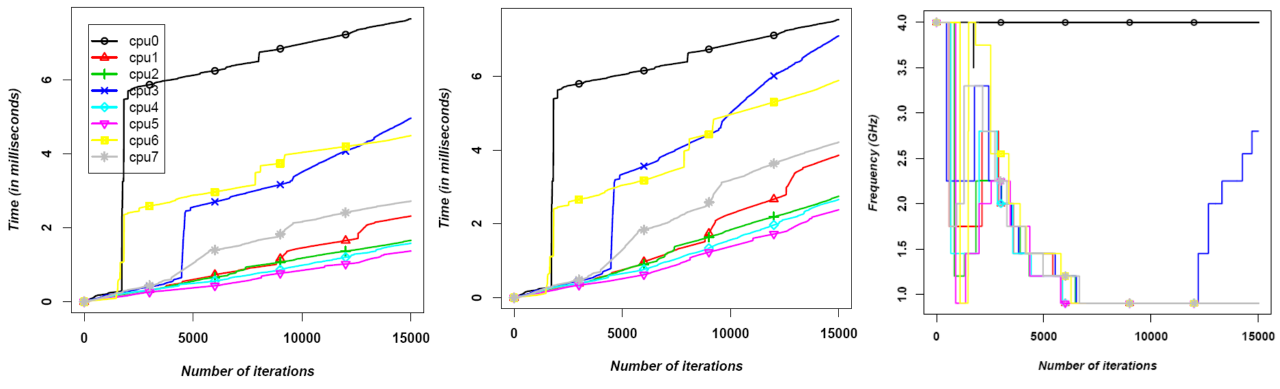


Figure 5 PageRank (sparse) on eight cores (a) runtime behavior of the baseline (b) runtime behavior after applying thread delaying (c) corresponding frequency level.

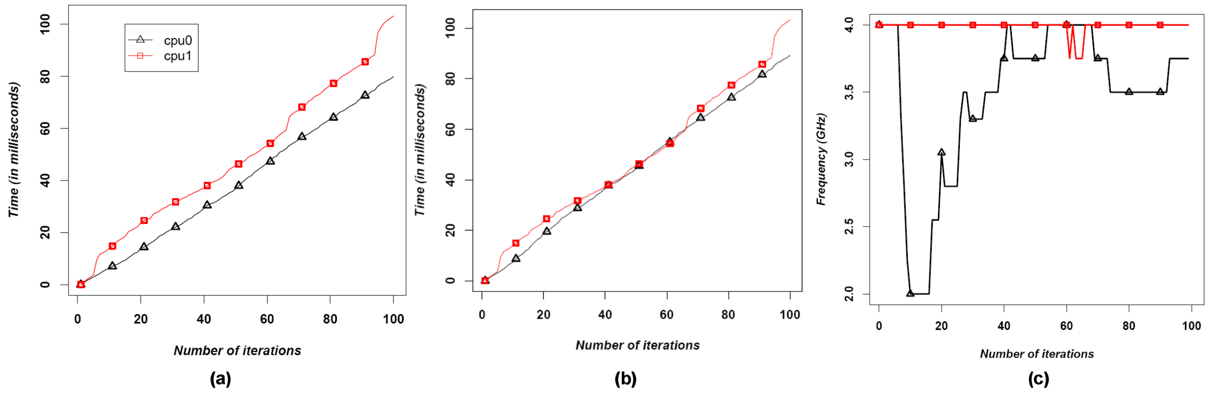


Figure 6 PageRank(lz77) on two cores (a) runtime behavior of the baseline (b) runtime behavior after applying thread delaying (c) corresponding frequency level.

observations are also obtained for the PageRank (sparse)'s 4p configuration and Summarization's 4p and 8p configurations.

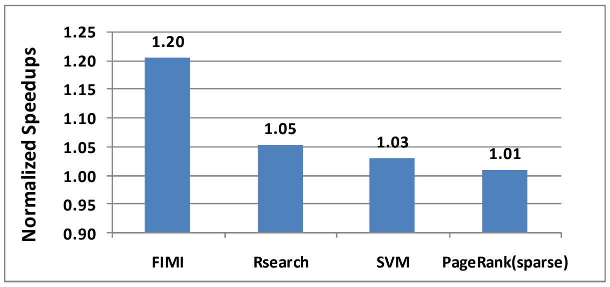
The effectiveness of thread delaying depends on whether the algorithm can adapt quickly at runtime; in other words, the algorithm chooses frequencies in a way that reflects the runtime behavior of the application. To demonstrate this, we use the example in Figure 6 (the same example used in Figure 1). In Figure 6 between iteration 10 and iteration 40, the time gap becomes smaller and smaller and our algorithm increments the frequency of the non-critical thread slowly. By doing that, the non-critical thread can avoid staying at a low frequency level for too long and becoming a false critical thread. If the non-critical 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 non-critical thread. The frequency of cpu1 (the critical thread) is slightly scaled down from 4 GHz to 3.75 GHz (see iterations between iterations 60-65). 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 in imbalanced workloads. Moreover, our thread delaying algorithm can also save a reasonable amount of energy for relatively balanced workloads. For example, Gauss is a balanced workload and it is hard to distinguish which threads are critical or non-critical (due to space constraints, we are not showing the graph). However, we still can achieve 6% energy savings without any performance penalty.

From above 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

Figure 7 shows the performance benefit of our thread balancing over the baseline for four RMS workloads. Performance benefit ranges from 1% to 20%. PageRank (sparse) shows huge imbalance during parallel execution and thus we have large amount of energy savings from thread delaying. However, thread balancing cannot give much performance improvement to PageRank. The reason is because of cache misses. Prioritizing the issue of the slow thread results in a shift in pipeline stalls from the issue stage to the backend of the in-order core because this benchmark suffers from a significant amount of load misses.



Benchmark Name	Opportunity(%)	Correction(%)
FIMI	30	100
Rsearch	24	49
SVM	56	100
PageRank (sparse)	4	1

Figure 7 Performance Results for Thread Balancing

Therefore the performance of the slow thread is not significantly improved.

Overall, the performance benefit correlates with imbalance levels. For example, in FIMI, there is a large level of thread imbalance and a corresponding amount of performance improvement by administering issue priority to the slower thread. We begin our analysis by determining the efficacy of this algorithm. We first present the opportunity, or the percentage of cycles that both threads have available instructions that are ready to be issued and a decision must be made between the threads. If the slow thread does not have available instructions to issue, then shifting priority to the slow thread will provide no benefit. FIMI and SVM have the most opportunity to give priority to the slow threads. Figure 7 also shows the correction of imbalance, which is defined to the percentage of the number of iterations that are caught up by the slow thread with our thread balancing method. As can be seen FIMI and SVM have 100% imbalance correction with this algorithm and are operating in an ideal situation.

6. RELATED WORK

There are some previous works related to thread delaying. Liu et al. [25] 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. [25] is that our approach runs at a finer grain, adapting to run-time 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 [25] cannot. Additionally our meeting point algorithm provides an opportunity to apply thread balancing on an SMT core. The scheme in [25] does not provide this opportunity.

The second work related to our work is called the thrifty barrier [24]. 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 [25]. 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 [1][22][23]. They created a performance-asymmetric multi-core 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 does not help to speed up the performance or save energy in this case. Therefore we are addressing the problem different from [1][22][23].

The DVFS algorithm can also be implemented at the operating system level. Lorch and Smith [26] 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 hardware targeting many-core system.

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 [6][12][17][20][32][34]. However, the focus is on prioritizing threads that are ready to execute, i.e. 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.

7. CONCLUSION

In this paper, we first present a novel mechanism called meeting point thread characterization, which dynamically estimates the criticality of the threads in a parallel execution. Knowing the criticality of each thread can be used in many different scenarios. In particular, we designed two novel schemes called thread delaying and thread balancing by using the thread criticality information in order to save energy and improve performance, respectively.

Thread Delaying combines per-core DVFS and meeting point thread characterization together to reduce energy consumptions on non-critical threads. Our experiments with several RMS applications have shown that this thread delaying mechanism is very effective. For example, for PageRank, which represents an important category of emerging applications such as Google's web search engine, the proposed mechanism can achieve up to more than 40% energy savings without any performance loss for four and eight-core configurations.

Thread balancing gives higher priority in the issue queue of an SMT core to the critical thread and by doing so, the overall performance of parallel regions can be improved. Our experiments have shown that our thread balancing mechanism can improve performance for various RMS workloads, ranging from 1% to 20%.

8. ACKNOWLEDGMENTS

This work has been partially supported by the Spanish Ministry of Education and Science under grants TIN2004-03702 and TIN2007-61763 and Feder Funds. We would like to thank the referees for their helpful comments and suggestions.

9. REFERENCES

[1] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 506–517, Washington, DC, USA, 2005. IEEE Computer Society

[2] OpenMP Architecture Review Board. Openmp application program interface, 2005.

[3] S. Y. Borkar. Platform 2015: Intel processor and platform evolution for the next decode. *Intel White Paper*, 2005

[4] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattcy7sh: A framework for architectural-level power analysis and optimizations. *ACM SIGARCH Computer Architec-ture News*, 28, 2000.

[5] T.D. Burd and R.W. Brodersen. Energy efficient cmos microprocessor design. System Sciences. *Proceedings of the Twenty-Eighth Hawaii International Conference*, 1995.

[6] Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and Enrique Fernandez. Dynamically controlled resource allocation in smt processors. *Microarchitecture*, 2004.

[7] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computer*, 22(4), 1973.

[8] P. Chaparro, J. Gonzalez, G. Magklis, Q. Cai, and A. Gonzalez. Understanding the termal implications of multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 18(8), 2007.

[9] T. Chelcea and S. M. Nowick. Robust interfaces for mixed-timing systems with application to latency-insensitive protocols. *Proceedings of the 38th Design Automation Conference*, 2001.

[10] Intel Corporation. Computer intensive, highly parallel application and uses. *Intel Technology Journal*, 9(2), 2005.

[11] Intel Corporation. Intel's tera-scale research prepares for tens, hundreds of cores, 2006.

[12] A. El-Moursy and D.H. Albonesi. Front-end policies for improved issue efficiency in smt processors. *High-Performance Computer Architecture*, 2003.

[13] S. Fischer. Technical overview of the 45nm next generation intel core microarchitecture (penryn), 2007.

[14] T. Fischer, J. Desai, B. Doyle, S. Naffziger, and B. Patella. A 90-nm variable frequency clock system for a power-managed itanium architecture processor. *IEEE Journal of Solid-State Circuits*, 41, 2006.

[15] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. Valentine. The intel pentium m processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2), 2003.

[16] P. Hazucha, T. Karnik, B.A. Bloechel, C. Parsons, D. Finan, and S. Borkar. Area-efficient linear regulator with ultra-fast load regulation. *Solid-State Circuits, IEEE Journal of*, 40, 2005.

[17] H. Homayoun, K.F. Li, and S. Rafatirad. Thread scheduling based on low-quality instruction prediction for simultaneous multithreaded processors. *IEEE-NEWCAS Conference*, 2005.

[18] Chenming Hu. Low-voltage cmos device scaling. *Solid-State Circuits Conference*, 1994.

[19] Anoop Iyer and Diana Marculescu. Power and performance evaluation of globally asynchronous locally synchronous processors. *ACM SIGARCH Computer Architecture News*, 30, 2002.

[20] R. Jain, C. Hughes, and S. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *23rd IEEE International Real-Time Systems Symposium*, 2002.

[21] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. *Micro, IEEE*, 25, 2005.

[22] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In MICRO 36: *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 81, Washington, DC, USA, 2003. IEEE Computer Society.

[23] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. *Proceedings of the 31st annual international symposium on Computer architecture*, Washington, DC, USA, 2004. IEEE Computer Society.

[24] J. Li, J.F. Martinez, and M.C. Huang. The thrifty barrier: energy-aware synchronization in shared-memory multiprocessors. *High Performance Computer Architecture*, 2004.

[25] C. Liu, A. Sivasubramaniam, M. Kandemir, and M.J. Irwin. Exploiting barriers to optimize power consumption of cmps. *Parallel and Distributed Processing Symposium*, 2005.

[26] Jacob R. Lorch and Alan Jay Smith. Improving dynamic voltage scaling algorithms with pace. *ACM SIGMETRICS*, 2001.

[27] G Magklis, P. Chaparro, J. Gonzalez, and A. Gonzalez. Independent front-end and back-end dynamic voltage scaling for a gals microarchitecture. *ISLPED*, 2006.

[28] G Magklis, J. Gonzalez, and A. Gonzalez. Frontend frequency-voltage adaptation for optimal energy-delay2. *International Conference on Computer Design*, 2004.

[29] Pedro Marcuello, Antonio Gonzlez, and Jordi Tubella. Speculative multithreaded processors. *Supercomputing*, 1998.

[30] T. Olsson, P. Nilsson, T. Meincke, A. Hemam, and M. Torkelson. A digitally controlled low-power clock multiplier for globally asynchronous locally synchronous designs. *ISCAS* 2000 Geneva.

[31] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *ACM SIGOPS Operating Systems Review*, 30, 1996.

[32] B. Robatmili, N. Yazdani, S. Sardashti, and M. Nourani. Thread-sensitive instruction issue for smt processors. *Computer Architecture Letters, IEEE*, 3, 2004.

[33] G. Semeraro, D. H. Albonesi, G. Magklis, M. L. Scott, S. Dropsho, and S. Dwarkadas. Hiding synchronization delays in a gals processor microarchitecture. *Proceedings of the 10th International Symposium on Asynchronous Circuits and Systems*, 2004.

[34] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. *ACM SIGARCH Computer Architecture News*, 24, 1996.

[35] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. Softsdv: A pre-silicon software development environment for the ia-64 architecture. *Intel Technology Journal,* 3(4), 1999.

[36] Q. Wu, P. Juang, M. Martonosi, and D.W. Clark. Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors. *High-Performance Computer Architecture*, 2005.

[37] W. Zhu, J. del Cuvillo, and G. R. Gao. Performance characteristics of openmp language constructs on a many-core-on-a-chip architecuture. *The 2nd International Workshop on OpenMP (IWOMP),* 2006.