Thread Shuffling: Combining DVFS and Thread Migration to Reduce Energy Consumptions for Multi-core Systems

Qiong Cai*, José González†, Grigorios Magklis*, Pedro Chaparro, Antonio González*
*Intel Barcelona Research Center, Intel Labs Barcelona - UPC (Spain)
{qiong.cai, griogorios.magklis, antonio.gonzalez}@intel.com
†Intel, Spain
{pepe.gonzalez}@intel.com

***Abstract*—In recent years, multi-core systems have become mainstream in computer industry. The design of multi-cores takes advantage of thread-level parallelism in emerging applications that are computationally intensive and highly parallel. Energy efficiency is one of the biggest challenges in the design of multi-core systems, and workload imbalance among parallel threads is one of sources of energy inefficiency. Many techniques based on dynamic voltage frequency scaling (DVFS) are proposed to save energy consumptions on multi-cores, but all of them assume that each core in a multi-core system contains only one hardware context and only one thread can execute on one core at a time. However, mainstream multi-core systems are moving to have simultaneous multi-threading (SMT) support in cores, and existing DVFS-based techniques are not effective to achieve maximum energy savings. In this paper, we present a novel technique called thread shuffling, which combines thread migration and DVFS to achieve maximum energy savings and maintain performance on a multi-core system supporting SMT. Thread shuffling is implemented and simulated in a cycle-accurate x86 multi-core system. The experiments show that it achieves up to 56% energy savings without performance penalty for selected Recognition, Mining and Synthesis (RMS) applications from Intel Labs.**

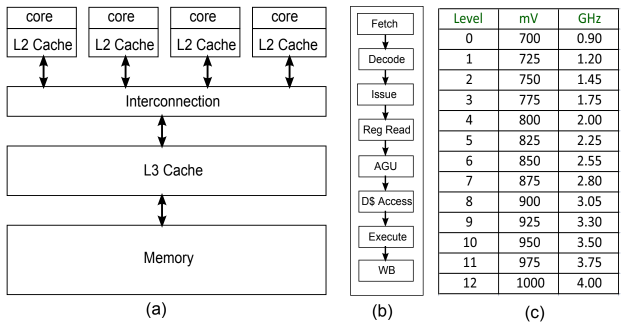
***Index Terms*—DVFS, thread migration, multi-core, thread shuffling, SMT**

I. Introduction

Pollack's rule states that performance increase is roughly proportional to square root of increase in complexity [4]. In another words, a single-threaded processor will provide a diminishing return in performance with respect to power and a multi-core microarchitecture can provide near linear performance improvement with respect to power and area [4]. In recent years, multi-core systems have become mainstream in computer industry [14], [15], [21]. The design of multi-core systems takes advantage of thread-level parallelism (TLP) to address of the problem of limited instruction-level parallelism (ILP) in serial applications. Moreover, many emerging applications are believed to be computationally intensive and highly parallel. For example, the recent single-chip cloud computer (SCC) [12] from Intel integrates 48 IA-32 cores to run highly parallel workloads such as Recognition, Mining and Synthesis (RMS) benchmarks [11].

Energy efficiency is one of the biggest challenges (probably the biggest challenge) faced by today's computer architects. Researchers found that workload imbalance among parallel threads is one of sources of energy inefficiency [6]. For example, in a fork-join execution model such as OpenMP [3], a group of parallel threads is generated at the fork-point of a parallel region and is synchronized at the join-point of the parallel region. In the ideal case, all threads reach this synchronization point called barrier at the same time. However, in a normal situation, some threads reach the barrier earlier than other threads. The faster threads go to a spin-loop and spend a large amount of time waiting for slower threads. During waiting time, the faster threads still consume energy, which leads energy inefficiency.

In order to reduce energy consumptions during waiting time, one approach is to put faster threads to sleep as soon as they reach the synchronization point and shut down the cores executing faster threads. To make this shutdown approach work, we need to make sure that the waiting time is long enough so that the energy saved in sleep mode pays off the energy wasted by putting the cores to sleep and waking them up. Another approach is to predict fast and slow threads at runtime and apply dynamic voltage and frequency scaling (DVFS) to cores executing fast threads. One representative technique in the second approach is thread delaying [6], which uses meeting point thread characterization to identify the critical thread (the slowest thread) of a multi-threaded application and the amount of slacks of non-critical threads (fast threads). The slack is defined to the amount of time a parallel thread can be delayed without impact on overall performance. The critical thread is the one with zero slack, and non-critical threads are those threads that could be delayed with no impact on performance.



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

Fig. 1. (a) Block diagram of a multi-core system (b) Pipeline of in-order SMT x86 core (c) Voltage-Frequency table

Thread delaying is effective and it can achieve energy savings up to more than 40% with negligible performance loss with respect to the shutdown approach for certain RMS benchmarks. However, it assumes that each core in a multi-core system contains only one hardware context and each core can only run one thread at a time. In this paper, we use hardware context and thread interchangeably. In current multi-core systems such as Nehalem from Intel [15] and POWER7 from IBM [14], each core supports simultaneous multithreading (SMT) and contains multiple hardware contexts. As we will demonstrate later in the paper, thread delaying cannot achieve maximum energy savings and maintain performance for multi-cores supporting SMT. For example, when running PageRank (one of RMS benchmarks) on 4 cores (each core contains two hardware contexts), thread delaying has 30% energy reduction but 14% performance loss. The same problem exists for all existing energy saving techniques based on DVFS for parallel applications, because all of them assume that each core contains only one hardware context.

In this paper, we propose a novel technique called thread shuffling, which takes core's SMT support into account. Similar to thread delaying, thread shuffling uses the meeting point thread characterization to predict thread criticality. Moreover, thread shuffling uses the concept of thread criticality degree of a thread and maps threads with similar criticality degrees into the same core through thread migration. Local DVFS is then applied to cores executing non-critical threads. By combining thread migration and DVFS, thread shuffling achieves great energy reduction and maintain performance.

In the rest of paper, we first describe our multi-core system with multiple clock domains in Section II. The baseline system supports local DVFS and thread migration. The meeting point thread characterization and thread delaying mechanisms are reviewed in Section III. We also show that thread delaying cannot achieve maximum energy savings and maintain performance on a multi-core system supporting SMT. Section IV describes our thread shuffling algorithm and its implementation in details. We show the simulation framework and performance results of thread shuffling in Section V. We also discuss the related work in Section VI. The paper concludes in Section VII.

II. A Multi-core System with Multiple Clock Domains

Figure 1(a) shows the block diagram of our multi-core system. The system consists of multiple Intel64/IA32 cores. The pipeline of in-order SMT core illustrated in Figure 1(b) is similar to Intel ATOM core 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 a reference clock signal as input 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 [8].

Each 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 [13], [20], [22] and some commercial designs [9]. We assume a limited range of voltages and frequencies, as shown in Figure 1(c).

III. Identification of Critical Threads and Thread Delaying

In this section, we first define what critical threads are in the context of parallel applications and review a mechanism called meeting point thread characterization that identifies critical threads at runtime. Thread delaying technique proposed by Cai et al [6] uses meeting point thread characterization to reduce energy consumptions on multi-core systems. We also show that thread delaying is not effective when cores in a multi-core system have SMT capability. The ineffectiveness of thread delaying is the motivation behind the design of our new algorithm called thread shuffling, which addresses inefficiency of thread delaying on SMT cores by combining thread migration and DVFS together.

A. *Identification of Critical Thread*

Meeting point thread characterization is used to identify the critical thread dynamically during program execution by checking the workload balance at intermediate points of a parallel section. These intermediate points are called meeting points. For a parallel loop, a natural location of a meeting point is at the back edge of the loop, because the back edge of a loop is visited many times by all

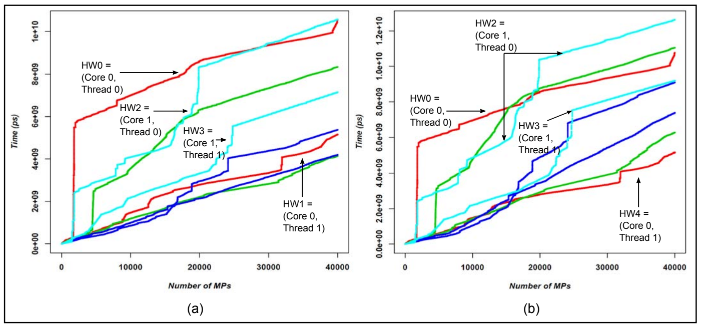


Fig. 2. Thread delaying cannot reduce energy and maintain performance at the same time when parallel threads run on multiple SMT cores. (a) The cumulative execution time of PageRank-sparse (a RMS benchmark) on four SMT cores. Each core contains two hardware contexts. Two threads running on the same core are illustrated with the same color. For purpose of discussion, four hardware contexts are specified by their core identifiers and thread identifiers. (b) The cumulative execution time of PageRank-sparse after applying thread delaying on four SMT cores.

threads at runtime. A special instruction mp_inst representing the meeting point is inserted into the last statement of the parallelized loop. The insertion of a meeting point can be done by the hardware, the compiler or the programmer.

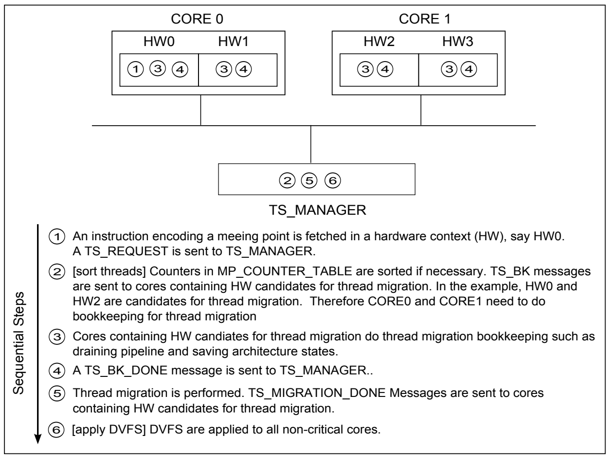
Every time a core decodes mp_inst, a thread-private counter located in the processor frontend is incremented. This counter is a proxy for the slack between the critical thread and a non-critical thread. A critical thread is the one with the smallest counters, and the slack of a thread can be estimated as the difference of its counter and counter of the slowest counter.

Thread delaying uses thread criticality to slow down fast threads by dynamically scaling down the voltage and frequency of cores running fast threads in such a way that the energy consumptions are reduced. At specific intervals of time, each core utilizes meeting point thread characterization to estimate the slack of a parallel thread running on the core. It computes the voltage and 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 delaying assumes that each core can only execute one thread at a time.

B. *Thread Delaying is Not Effective for SMT Cores*

Thread delaying, however, is not effective when parallel threads run on multiple SMT cores. The ineffectiveness of thread delaying is illustrated in Figure 2. Figure 2(a) shows the run time behavior of the hottest region in PageRank-sparse (a RMS benchmark). The hottest region is a parallel loop. In this example, the baseline system contain four SMT cores. Each core has two hardware contexts and allows two threads to run at the same time. The parallel loop is partitioned into eight threads, which are mapped to eight hardware contexts at run time. The x-axis in the figure represents the number of times a thread visits a meeting point. The y-axis represents the cumulative execution time of each thread. Two threads running on the same core are illustrated with the same color. For purpose of discussion, text labels are used to annotate four hardware contexts: HW0, HW1, HW2 and HW3. Each hardware context is specified by its core identifier and thread identifier. For example, HW0 denotes a hardware context in core 0 and thread 0.

From Figure 2(a), we can see that HW0 is the critical hardware context most of time and HW1 is a non-critical hardware context all the time. If the core contains only one hardware context, then the gap between HW0 and HW1 can be reduced after thread delaying



① An instruction encoding a meeing point is fetched in a hardware context (HW), say HW0. A TS_REQUEST is sent to TS_MANAGER.
② [sort threads] Counters in MP_COUNTER_TABLE are sorted if necessary. TS_BK messages are sent to cores containing HW candidates for thread migration. In the example, HW0 and HW2 are candidates for thread migration. Therefore CORE0 and CORE1 need to do bookkeeping for thread migration
③ Cores containing HW candiates for thread migration do thread migration bookkeeping such as draining pipeline and saving architecture states.
④ A TS_BK_DONE message is sent to TS_MANAGER..
⑤ Thread migration is performed. TS_MIGRATION_DONE Messages are sent to cores containing HW candidates for thread migration.
⑥ [apply DVFS] DVFS are applied to all non-critical cores.

Fig. 3. Overview of thread shuffling algorithm. The above steps are sequential. Each step can start only when the above step is finished. For example, step 5 (thread migration in the manager) can be executed only when messages sent from all affected hardware contexts are received by the thread shuffling manager. Step 2 (sort threads) and step 6 (apply DVFS) will be described in details later.

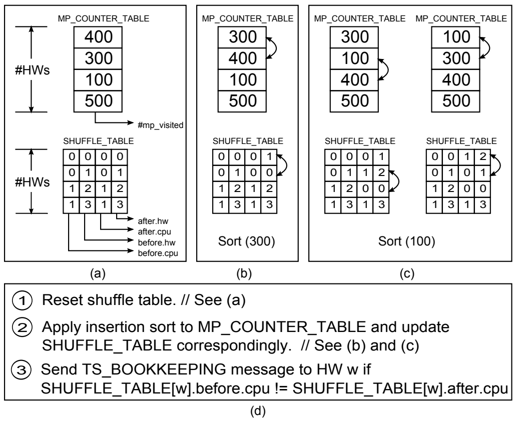
is applied. The gap between the critical thread and a non-critical thread represents potential energy savings. However, in this case, HW0 and HW1 run on the same core. Figure 2(b) shows that the gap between HW0 and HW1 stays the same after thread delaying is applied. There is no any energy savings from HW1, even though HW1 is non-critical all the time. There are two reasons why thread delaying cannot achieve optimal energy savings on SMT cores. The first reason is that thread delaying assumes each core contains only one hardware context. The second reason is that local DVFS is applied at the core level instead of thread level.

Another observation from the figure is that it is difficult to recover performance loss when the aggressive voltage and frequency scaling is applied at the beginning of execution time. For example, the slack between HW0 and HW2 is large at the beginning and aggressive DVFS is applied. However, the slack between HW0 and HW2 suddenly becomes very small. Under this situation, it is difficult for thread delaying to react and change the voltage and frequency level of core 1 back to the maximal one. This causes big performance loss illustrated in Figure 2(b).

Both non-optimal core-level DVFS and aggressive DVFS problems will be addressed by our proposed technique called thread shuffling. Thread shuffling combines thread migration and conservative DVFS together to reduce more energy consumptions and maintain performance. For the PageRank example illustrated in Figure 2, thread delaying has 30% energy reduction but 14% performance loss, whereas thread shuffling obtains 50% energy savings without any performance loss.

IV. Thread Shuffling and its Implementation

The idea of thread shuffling is to map threads with similar criticality degrees into the same core through thread migration and then apply dynamic voltage and frequency scaling to cores containing non-critical threads. In meeting point thread characterization, each thread has a counter to accumulate the number of times a meeting point is visited, and the criticality of a thread is approximated as the difference between its own counter and the counter of the slowest one. We approximate the criticality degree of a thread as the position of its own counter in a reverse sorted sequence of all counters. For example, if the counter values are 200, 100, 300, 400 for thread 0,



① Reset shuffle table. // See (a)
② Apply insertion sort to MP_COUNTER_TABLE and update SHUFFLE_TABLE correspondingly. // See (b) and (c)
③ Send TS_BOOKKEEPING message to HW w if SHUFFLE_TABLE[w].before.cpu != SHUFFLE_TABLE[w].after.cpu

Fig. 4. Sorting threads. (a) examples of MP_COUNTER_TABLE and SHUFFLE_TABLE. SHUFFLE_TABLE is reset whenever sorting thread is called. (b)-(c) steps of insertion sort of counters in MP_COUNTER_TABLE. Entries (fields after.cpu and after.hw) in SHUFFLE_TABLE are updated accordingly. (d) the algorithm of sorting threads.

```
1. critical_core = 0;
2. ref_counter = find_largest_mp_counter (critical_core)
3. for (i = 1; i < TOTAL_NUM_CORES; i++)
4.    cmp_counter = find_smallest_mp_counter (i);
5.    scale = ref_counter / cmp_counter;
6.    freq_level = map_scale_to_frequency_level (scale);
7.    freq_level = update_frequency_history_table(i, freq_level);
8.    change_frequency_VDD (FREQ[freq_level], VDD[freq_level]);
```

Fig. 5. Algorithm of DVFS in Thread Shuffling

1, 2, 3, respectively, then thread 1 is the most critical thread with degree 3, thread 2 has criticality degree of 2, and thread 3 is the most non-critical thread with degree of 0.

Thread shuffling addresses the problem of non-optimal DVFS in thread delaying by first grouping threads with similar criticality degrees into the same core and then apply DVFS. For example, HW0 and HW2 in Figure 2(a) will be mapped into the same core at runtime. This remapping increases chance of reducing the gap between HW0 and HW2 and thus the energy reduction increases.

A. Overview Algorithm

Figure 3 shows the overview of our thread shuffling algorithm. The major part of algorithm is implemented in a centralized hardware manager called TS_MANAGER (thread shuffling manager), which coordinates jobs between hardware contexts and itself and applies thread migration and DVFS to hardware contexts and cores, respectively.

The algorithm illustrated in Figure 3 is self-explained. One additional remark is that after the manager receives a shuffling request TS_REQUEST from a hardware context, it checks whether the following condition is true or not:

```
(current_cycle - last_config_cycle)
      > MAX_CONFIG_INTERVAL
```

where current_cycle is the current cycle when the manager receives the request, last_config_cycle is the cycle when the last shuffling is performed and MAX_CONFIG_INTERVAL is a parameter to adjust the frequency of thread shuffling. In this study, MAX_CONFIG_INTERVAL is set to five millions cycles, since our thread shuffling is fine-grained and lightweight enough to be implemented in hardware. If the above condition is true, then the manager starts a new thread shuffling.

B. Sorting Threads and Thread Migration

In Figure 3(a)-(c) illustrates the data structures and algorithms of sorting algorithm. After sorting is done, the manager sends a TS_BK message to a hardware context hw if the the following condition is true.

```
    SHUFFLE_TABLE[hw].before.cpu !=
          SHUFFLE_TABLE[hw].after.cpu
```

For example, in Figure 4(c), SHUFFLE_TABLE shows that hardware context 0 in cpu 0 will be replaced by hardware context 2 in cpu 1. Therefore, the manager sends a TS_BK message to core 0.

The sorting algorithm is summarized in Figure 4(d). One additional remark is that SHUFFLE_TABLE is also used in thread migration step. For example, Figure 4(c) shows that hardware context 0 and hardware context 2 need to be swapped to make the counters in MP_COUNTER_TABLE sorted. This step is performed during thread migration when thread migration bookkeeping is finished.

C. Local DVFS on non-Critical Cores

Besides sorting threads and thread migration, another important part of thread shuffling algorithm is to choose suitable voltage and frequency levels for cores containing non-critical threads. Figure 5 shows the DVFS algorithm used in thread shuffling.

For each core containing non-critical threads, the algorithm first computes a scaling factor based on two counter ref_counter and cmp_counter (step 5 in the figure). There are several ways to compute ref_counter and cmp_counter. As we discussed in Section III-B, it is difficult to recover performance loss if aggressive voltage and frequency scaling is applied. Therefore, we use conservative approach to compute these two counter values. Counter ref_counter is the largest counter value in core 0. Core 0 is called critical core, because it always contains the most critical threads after thread migration. For example, in Figure 4, ref_counter is equal to 300. Other cores except core 0 are called non-critical cores. Counter cmp_counter is the smallest counter value in a non-critical core. In Figure 4, cmp_counter is equal to 400 for core 1. Thus, the scaling factor for core 1 is $300/400 = 0.75$. In our current model, voltage scaling is implemented as a discrete function with 13 frequency levels shown in Figure 1(c). Therefore, the scaling factor 0.75 is multiplied by 13 (step 6), which gives us level 10 after rounding.

After the frequency level f is obtained, the internal HISTORY_TABLE in the manager is updated properly (step 7). Each entry of the table contains a two bit up-down saturating counter. If the frequency level f is k, then entry k is incremented and every other entry is decremented. The final frequency level for the next interval is the one with the largest counter in the HISTORY_TABLE. The purpose of HISTORY_TABLE is used to reduce the effect of temporal noise in the estimation of the slack [6].

The final step of DVFS algorithm (step 8) accesses voltage-frequency table in Figure 1(c) based on the computed frequency level. For example, level 10 is computed for core 1. Therefore, voltage 0.95V and frequency 3.5GHz will be selected for core 1 in next interval of time.

V. Experiments

The simulation framework used in our thread shuffling evaluation contains a full system functional simulator and an x86 cycle-accurate performance simulator. SoftSDV [23] for Intel64/IA32 processors is our functional simulator, which can simulate not only multithreaded primitives including locks and synchronization operations but also shared memory and events. Redhat 3.0 EL is booted as the guest operating system in SoftSDV. In all of our simulation, only less than 1% of simulated instructions are from the operating system and thus the impact of the operating system is negligible.

Parameter	Value
Processor	In-order x86 core, 2-way SMT
L1 Instruction Cache (private)	32KB, 4-way, 64B
L1 Data Cache (private)	32KB, 4-way, 64B
L2 Cache (unified and private)	512KB, 16-way, 64B
L3 Cache (unified and shared)	8MB, 16-way, 64B
Memory	always hit, 500 cycles access penalty
Network Protocol	MESI

(a)

Benchmark	Application Domain
PageRank-lz77	Search Engine
PageRank-sparse	Search Engine
Rsearch	Bioinformatics
Summarization	Text Data Mining

(b)

Fig. 6. (a) The architectural parameters (b) The RMS benchmarks used in our evaluation.

Benchmarks	Thread Delaying vs Baseline		Thread Shuffling vs Baseline		Thread Shuffling vs Thread Delaying	
	Execution Time	Energy Consumption	Execution Time	Energy Consumption	Execution Time	Energy Consumption
PageRank-lz77	1.00	0.93	0.99	0.90	0.99	0.97
PageRank-sparse	1.14	0.71	0.98	0.49	0.86	0.69
Rsearch	1.01	0.97	0.98	0.97	0.97	1.00
Summarization	1.00	0.82	1.00	0.76	1.00	0.93

(a)

Benchmarks	Thread Delaying vs Baseline		Thread Shuffling vs Baseline		Thread Shuffling vs Thread Delaying	
	Execution Time	Energy Consumption	Execution Time	Energy Consumption	Execution Time	Energy Consumption
PageRank-lz77	1.02	0.93	1.02	0.93	1.00	1.00
PageRank-sparse	1.29	0.65	0.99	0.50	0.77	0.77
Rsearch	1.02	0.94	0.96	0.92	0.94	0.98
Summarization	1.00	0.79	1.00	0.73	1.00	0.92

(b)

Fig. 7. Performance results of thread shuffling (a) on 4 cores (b) on 8 cores. The baseline configuration is that whenever a thread reaches the barrier, the thread is put into sleep mode and consumes zero power.

The functional simulator feeds Intel64/IA32 instructions into an x86 performance simulator. The performance simulator uses a power model based on activity counters and energy per access, similar to Wattch [5]. In our evaluation, the energy includes dynamic, idle and leakage energy. The baseline is very aggressive. It assumes that every core is running at full speed and stops when it is completed. Once the core stops, it consumes zero power.

Thread shuffling is implemented in-house cycle-accurate performance simulator for x86 multi-core system illustrated in Figure 1(a). For performance comparison, thread delaying is also implemented in the same performance simulator. Multiple clock domains explained in Section II are faithfully implemented in the simulator. Each core is an in-order SMT core with the pipeline shown in Figure 1(b). The simple in-order core is low power and is suitable for a many-core chip. The detailed architectural parameters are shown in Figure 6(a).

A. Benchmarks

The Recognition, Mining, and Synthesis (RMS) benchmarks from Intel are a set of emerging multithreaded applications for Tera-scale systems [11]. The RMS benchmarks are highly compute-intensive and highly parallel applications including data mining on text and media, bio-informatics and search engine.

From the RMS benchmark suite, we have chosen those benchmarks that clearly show workload imbalance. These benchmarks are shown in Figure 6(b). 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 format or a compressed version. The compression is a simplified LZ77-based method. Rsearch is used in bioinformatics to search a homologous RNA in a database. The benchmarks were developed by expert programmers and parallelized by hand to achieve maximal scalability. 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 version and then selecting the hottest region, which normally is a parallel loop. For all of the benchmarks, the selected parallel regions represent almost 99% of total execution time. 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 are executed before a simulation is terminated.

B. *Performance Results*

Figure 7 shows the performance results of thread shuffling with respect to the baseline configuration and thread delaying on 4 and 8 cores, respectively. Under the baseline configuration, whenever a thread reaches the barrier, the thread is put into sleep mode and consumes zero power. There are three observations from the results. First, thread shuffling can reduce more energy consumptions than thread delaying. On 4 cores, thread shuffling can obtain up to 51% energy savings with respect to the baseline and achieve 31% energy savings with respect to thread delaying. Second, thread shuffling is robust and does not cause any performance slowdown across benchmarks, whereas thread delaying has 14% and 29% slowdowns with respect to the baseline on 4 cores and 8 cores, respectively. Third, thread shuffling is scalable. Performance in terms of execution time and energy consumption with respect to the baseline and thread delaying is maintained when the number of cores is increased from 4 to 8, or the number of hardware contexts is increased from 8 to 16.

C. *Three Configurations of Thread Shuffling*

Our DVFS algorithm is conservative. When computing the scaling factor (step 5 in Figure 5), the largest counter value in the critical core and the smallest counter value in a non-critical core are chosen for ref_counter and cmp_counter, respectively. By doing this, the scaling factor ref_counter/cmp_counter tends to be close to one. The motivation behind this is that it is difficult to recover performance loss when the scaling factor is small and DVFS is applied too aggressively.

A SMT core contains several hardware contexts and each hardware context has one counter approximating thread criticality. For a sequence of counters in a SMT core, there are many choices for ref_counter and cmp_counter, which lead to many combinations of ref_counter and cmp_counter. In this study, we use three simplest statistics methods, namely, maximum, minimum, and average, to find ref_counter (or cmp_counter) in a given sequence of counters. Therefore, there are nine possible combinations of ref_counter and cmp_counter. However, from our experiments, we found that the performance loss is big (more than 20% for some benchmarks) when maximum and average methods are used to choose cmp_counter. Thus, only minimum method is used when choosing cmp_counter value. Figure 8(a) shows three combinations of ref_counter and cmp_counter. The default configuration of thread shuffling uses the combination of minimum of counters in the critical core and maximum of counters in a non-critical core. The aggressive configuration of thread shuffling uses the combination of maximum of counters in the critical core and maximum of counters in a non-critical core. The third configuration called midpoint version of thread shuffling uses the combination of average value of counters in the critical core and maximum of counters in a non-critical core.

Figure 8(b) shows the performance results of three configurations of thread shuffling. We can see that the aggressive version achieves the best energy reduction. For PageRank-sparse, it reduces up to 57% energy consumption with respect to the baseline. On other hand, the aggressive verson has 6% performance slowdown for PageRank-sparse. For other three benchmarks, the performance loss is negilible for the aggressive version. The midpoint version of thread shuffling achieves the best balance between the performance loss and energy consumption reduction. It achieves energy consumption reduction up to 56% with neglible performance penality.

VI. Related Work

The closest work related to thread shuffling is thread delaying [6]. Both techniques use the meeting point thread characterization to identify critical and non-critical threads. Thread delaying uses information about thread criticality to apply local DVFS to cores containing non-critical threads. Thread delaying assumes each core contains only one hardware context. As we demonstrated in previous sections, thread delaying cannot achieve maximum energy savings and maintain performance when cores in a multi-core system support multiple hardware contexts (or SMT support). Thread shuffling is designed to address the ineffectiveness of thread delaying on SMT cores. It introduces the concept of criticality degree of a thread. Thread shuffling maps threads with similar criticality degrees into the same core by thread migration and then applies DVFS to non-critical cores. By combining thread migration and DVFS, thread shuffling achieves the maximum energy savings without performance penalty.

All existing energy savings techniques based on DVFS for parallel applications [2], [6], [18] assume that cores in a multi-core system do not have SMT support. However, current mainstream multi-core systems such as Nehalem from Intel [15] and Power7 from IBM [14] have SMT support. It is necessary to take SMT support into account when a energy saving technique is designed. We are not aware that any existing technique combines DVFS and thread migration to reduce energy consumptions on SMT cores.

In thread shuffling, meeting point thread characterization is used because its simplicity and effectiveness for parallel sections. However, it assumes that the parallel section is statically scheduled [6]. Another thread characterization technique called thread criticality predictor (TCP) [2] does not have this constraint and it demonstrated that TCP worked well for benchmarks containing loops with variable iteration times. In our future work, we may combine TCP and thread shuffling together to reduce energy consumption for even wider ranges of applications.

Similar to thread delaying, thread shuffling is motivated by the workload imbalance among parallel threads. This type of performance asymmetry due to workload imbalance is different from the performance asymmetry exploited in the literature [1], [16], [17]. Their approach used a performance-asymmetric multi-core system, which includes high performance and complex big cores to execute serial code and many simple cores to execute parallel code. However, in our case, the asymmetry comes from the workload imbalance among parallel threads from the same parallel region. We are addressing different performance-asymmetry problems.

DVFS and thread migration can be implemented at software level [10]. The software level can be operating system level or application level. Existing software level energy saving techniques based on DVFS [19], [24] schedule a task in such a way that the frequency of a CPU is scaled down to save energy and meet the deadline of the task.

ref_counter			cmp_counter			Configuration
max	min	average	max	min	average	
	×		×			thread shuffling (default)
×			×			thread shuffling (aggressive)
		×	×			thread shuffling (midpoint)

(a)

Benchmarks	Thread Shuffling		Thread Shuffling-Aggressive		Thread Shuffling-Midpoint	
	Execution Time	Energy Consumption	Execution Time	Energy Consumption	Execution Time	Energy Consumption
PageRank-lz77	0.99	0.90	1.03	0.87	1.03	0.87
PageRank-sparse	0.98	0.49	1.06	0.43	1.01	0.44
Rsearch	0.98	0.97	1.00	0.95	0.99	0.95
Summarization	1.00	0.76	1.02	0.71	1.01	0.70

(b)

Fig. 8. (a) Three configurations of thread shuffling (b) Performance results of thread shuffling and its variants with respect to the baseline on 4 cores.

Software level scheduling algorithms are coarse-grained and target for only one CPU. Thread shuffling is fine-grained and lightweight enough to be implemented in hardware targeting multi-core systems. However, we believe that the idea of thread shuffling can be extended to software level. In our future work, we will investigate software level techniques based on DVFS and thread migration for reducing energy consumptions on multi-core systems supporting SMT.

VII. Conclusion

In this paper, we first show that thread delaying, one of the most advanced energy savings techniques for parallel applications, is not effective on a multi-core system when cores have SMT support. This problem exists in all existing energy savings techniques based on DVFS for parallel application, because they assume that each core in a multi-core system contains only one hardware context and allows only one thread to run at a time.

We present a novel technique called thread shuffling to address this problem. Thread shuffling uses both concepts of thread criticality and thread criticality degree. It dynamically maps threads with similar criticality degrees into the same core and then applies DVFS to non-critical cores. Our experiments with several RMS applications have shown that thread shuffling is very effective. For example, for PageRank, which represents an important category of emerging applications such as Google's web search engine, thread shuffling (midpoint version) can obtain up to 56% energy savings with respect to the baseline and achieve 38% energy savings with respect to thread delaying without any performance loss on 4 cores. Similar performance is obtained when the experiment is done on 8 cores.

References

[1] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 506–517, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Abhishek Bhattacharjee and Margaret Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 290–301, New York, NY, USA, 2009. ACM.

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

[4] Shekhar Borkar. Thousand core chips: a technology perspective. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 746–749, New York, NY, USA, 2007. ACM.

[5] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94, New York, NY, USA, 2000. ACM.

[6] Qiong Cai, José González, Ryan Rakvic, Grigorios Magklis, Pedro Chaparro, and Antonio González. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 240–249, New York, NY, USA, 2008. ACM.

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

[8] 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.

[9] 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.

[10] Mohamed Gomaa, Michael D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging smt and cmp to manage power density through the operating system. In *ASPLOS-XI*, pages 260–270, New York, NY, USA, 2004. ACM.

[11] Intel. Computer intensive, highly parallel applications and uses. 2005.

[12] Intel. A 48-core ia-32 message passing processor with dvfs in 45nm cmos. In *to appeared in ISSCC*, 2010.

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

[14] Ronald Kalla and Balaram Sinharoy. Power7: Ibm's next generation server processor. In *Hot Chip*, 2009.

[15] Rajesh Kumar and Glenn Hinton. A family of 45nm ia processors. In *IEEE International Solid-State Circuits Conference (ISSCC)*, 2009.

[16] 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.

[17] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 64, Washington, DC, USA, 2004. IEEE Computer Society.

[18] Chun Liu, Anand Sivasubramaniam, Mahmut Kandemir, and Mary Jane Irwin. Exploiting barriers to optimize power consumption of cmps. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 5.1, Washington, DC, USA, 2005. IEEE Computer Society.

[19] Jacob Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with pace. In *ACM SIGMETRICS*, 2001.

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

[21] Stefan Rusu, Simon Tam, Harry Muljono, Jason Stinson, David Ayers, Jonathan Chang, Raj Varada, Matt Ratta, and Sailesh Kottapalli. A 45nm 8-core enterprise xeon processor. In *ISSCC*, 2009.

[22] 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.

[23] 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.

[24] Qiang Wu, Margaret Martonosi, Douglas W. Clark, V. J. Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *MICRO 38*, pages 271–282, Washington, DC, USA, 2005. IEEE Computer Society.