J. Parallel Distrib. Comput. 72 (2012) 353–361

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Efficient and scalable scheduling for performance heterogeneous multicore systems ☆

Pengcheng Nie, Zhenhua Duan *

Institute of Computing Theory and Technology, and ISN Laboratory, Xidian University, No.2, South Taibai Road, Xi'an, China

ARTICLE INFO

Article history:
Received 31 August 2010
Received in revised form
6 December 2011
Accepted 14 December 2011
Available online 22 December 2011

Keywords:
Performance heterogeneous multicore
Scheduling
Algorithm
Operating systems

ABSTRACT

Performance heterogeneous multicore processors (HMP for brevity) consisting of multiple cores with the same instruction set but different performance characteristics (e.g., clock speed, issue width), are of great concern since they are able to deliver higher performance per watt and area for programs with diverse architectural requirements than comparable homogeneous ones. However, such power and area efficiencies of performance heterogeneous multicore systems can only be achieved when workloads are matched with cores according to both the properties of the workload and the features of the cores.

Several heterogeneity-aware schedulers were proposed in the previous work. In terms of whether properties of workloads are obtained online or not, those scheduling algorithms can be categorized into two classes: online monitoring and offline profiling. The previous online monitoring approaches had to trace threads' execution on all core types, which is impractical as the number of core types grows. Besides, to trace all core types threads have to be migrated among cores, which may cause load imbalance and degrade the performance. The existing offline profiling approaches profile programs with a given input set before really executing them and thus remove the overhead associated with the number of core types. However, offline profiling approaches do not account for phase changes of threads. Moreover, since the properties they have collected are based on the given input set, those offline profiling approaches are hard to adapt to various input sets and therefore will drastically affect the program performance.

To address the above problems in the existing approaches, we propose a new technique, ASTPI (Average Stall Time Per Instruction), to measure the efficiencies of threads in using fast cores. We design, implement and evaluate a new online monitoring approach called ESHMP, which is based on the metric. Our evaluation in the Linux 2.6.21 operating system shows that ESHMP delivers scalability while adapting to a wide variety of applications. Also, our experiment results show that among HMP systems in which heterogeneity-aware schedulers are adopted and there are more than one LLC (Last Level Cache), the architecture where heterogeneous cores share LLCs gain better performance than the ones where homogeneous cores share LLCs.

© 2012 Elsevier Inc. All rights reserved.

1. Introduction

With the development of the semiconductor technology, billions of fast transistors are pushed onto a single chip. The problem of interconnect delay and design complexity has become obvious [11]. Multicore architecture, also known as chip multiprocessors (CMPs), which include several processors on a single chip, are being widely touted as a solution to thermal and power problems by exploring thread level parallelism (TLP) [5]. In order to fully utilize these cores provided by CMP architecture, programs are designed to be multi-threaded. However, the computing characteristics of threads and programs often demonstrate substantial diversity. For example, the CPU-boundedness across different threads may vary widely. Even the characteristics of a single thread may vary during different phases of its execution [15]. Performance heterogeneous multicore processors, which consist of multiple cores with the same instruction set but different performance characteristics (e.g., clock speed, issue width), better accommodate such diversity with smaller die area and lower power consumption as compared to homogeneous ones [9,3,7,8,10,12]. Therefore, performance heterogeneous multicore architecture are gaining more and more attention. In this work, we focus on performance heterogeneous multicore systems, where cores differ in clock frequency and in the last-level-cache (LLC) sharing mode, because such

☆ This research is supported by the National Program on the Key Basic Research Project of China (973 Program) Grant No. 2010CB328102, the National Natural Science Foundation of China under Grant Nos. 60910004, 60873018, 91018010, 61133001, 61003078 and 61003079, SRFDP Grant 200807010012, and the ISN Lab Grant No. ISN1102001.

* Corresponding author.
E-mail addresses: pengcheng.nie@gmail.com (P. Nie), zhhduan@mail.xidian.edu.cn (Z. Duan).

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2011.12.005

systems can be easily emulated using existing multicore processors and are expected to play a promising role in future heterogeneous systems [14].

Power and area efficiencies of performance heterogeneous multicore systems can only be accomplished when workloads are matched with cores according to the properties of the workload and the features of the cores. For example, in a performance heterogeneous multicore system with several fast and powerful cores (high frequency, superscalar) and several simple and slow cores, if CPU-bound threads were assigned to slow cores and memory-bound threads to fast cores, fast cores would spend most of their time waiting and slow cores would be as busy as a bee, hurting system performance greatly. Hence, the matching mechanism must take into account the heterogeneity of the system and the workload, and, especially, the varying behavior of the threads over time. The matching job is typically done by a heterogeneity-aware scheduling algorithm in the operating system [14]. Generally, a heterogeneity-aware scheduling algorithm consists of three procedures, collecting properties of workloads, analyzing collected properties and scheduling threads to cores. In terms of whether properties of workloads are obtained online or not, heterogeneity-aware scheduling algorithms can be categorized into two classes: online monitoring [9,3] and offline profiling [14,13].

Existing online monitoring policies, say *IPC-Driven* (IPC stands for Instructions Per Cycle, which is a technique for measuring a thread's execution rate) algorithm proposed by Becchi and Crowley [3], periodically observe runtime behaviors of the running threads on each core type and assign threads to cores based on their relative speedup on different core types [9,3]. The thread that has the greatest fast-to-slow core speedup ratio has the highest priority to run on the fastest core. Though those online monitoring approaches adapt to heterogeneity of threads well, as the number of cores (and core types) on the chip increases [1,4], the overhead of performance monitoring grows and they become too time-consuming and impractical. Namely, previous online monitoring algorithms are *unscalable*. In order to observe runtime behavior, IPC-Driven algorithm periodically performs monitoring on all core types, which means the demands for different core types are the same. This sampling process, however, may cause *load imbalance* and hurt performance if one core type has more cores than another. That is, some cores have more threads than the others. Especially, the less the fast cores than the slow cores, the more threads will run on any fast core for monitoring purpose. Unlike online monitoring policies, offline profiling approaches, such as *HASS* (short for Heterogeneity-Aware Signature-Supported scheduling algorithm) presented by Shelepov et al. [14], profile programs with a given input set before really executing them and thus remove the overhead associated with the number of core types. However, offline profiling approaches like HASS do not account for phase changes of threads. Moreover, as the properties they are collected are based on the given input set, those offline profiling approaches are hard to adapt to various input sets and thus drastically affect program performance. In other words, offline profiling approaches provide simplicity and scalability at the cost of *sacrificing accuracy*. Furthermore, it requires cooperation from developers to perform profiling step.

In this paper, we propose ESHMP, an Efficient and Scalable HMP scheduling algorithm that delivers both accuracy and scalability. This approach is based on a new technique ASTPI (Average Stall Time Per Instruction), which is the average time that a thread spends in waiting for memory accesses for executing one instruction. For a thread, its average stall time on each core type could be the same, since each core type has the same memory hierarchy (see Fig. 1). Besides, we find out that ASTPI reflects a thread's efficiency in using fast cores. That is, if a thread has less average stall time then the core on which the thread executes will stall itself less. Thus it is sufficient to measure ASTPI on any core to estimate its performance on all core types. By comparing the relative benefits for different threads, the scheduler decides which thread is the best candidate for a particular core type. Moreover, in response to phase changes of applications, ESHMP periodically samples their behavior and makes thread-to-core assignment accordingly.

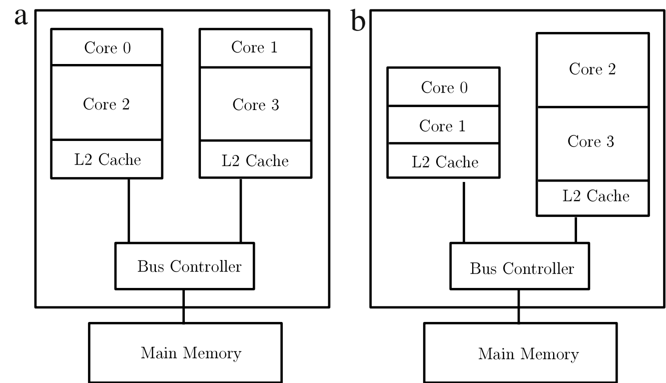


Fig. 1. Typical architecture of HMPs, where each core has the same memory hierarchy. Every two heterogeneous cores share a LLC in architecture (a) and every two homogeneous cores share a LLC in architecture (b).

The main contributions of this work can be summarized as follows:

- By monitoring ASTPI for a thread on only one core type instead of on all core types, our algorithm overcomes the unscalability and load imbalance in the existing online monitoring scheduling algorithms. In addition, the monitoring process is a periodic one, thus phase changes are considered in our approach. Moreover, our approach is based on runtime behaviors of threads, not on demands associated with a specific input set, which means our algorithm does not rely on input sets (see Section 3). Thus, our algorithm removes the inaccuracies existing in offline profiling algorithms.
- We formally prove that the stall time of a thread reflects its efficiency in using a fast core, i.e., the speedup of a thread on fast core relative to slow core is monotonically decreasing in the stall time of the thread (see Section 3.2).
- By experiments, we discover that among HMP systems in which heterogeneity-aware schedulers are adopted and there are more than one last-level-cache (LLC), the architecture where heterogeneous cores share LLCs gain better performance than the ones where homogeneous cores share LLCs (see Section 4). Also, the discovery is used in turn to evaluate that the ESHMP scheduler is more efficient than others.

We implemented ESHMP in the Linux 2.6.21 operating system and evaluated it on real multicore hardware where heterogeneity is emulated by setting the cores to run at different frequencies via DVFS (Dynamic Voltage and Frequency Scaling, a facility offered by most modern processors). We used CPU-bound scientific applications, and constructed workloads containing various applications: CPU-intensive and memory-intensive, single-phased and multi-phased.

We compared ESHMP with several other heterogeneity-aware schedulers including IPC-Driven algorithm and HASS. We found that for workloads consisting exclusively of single-phased applications, HASS performs well. It has nothing to do with the number of core types, but this algorithm is ineffective for workloads containing multi-phased applications. Conversely, IPC-Driven method is effective for workloads containing multi-phased applications, but its overhead grows drastically as the number

of core types increases. ESHMP, on the other hand, effectively addresses both problems. Its time complexity is constant in the size of core types. Moreover, it accommodates both single-phased and multi-phased workloads well. The greatest benefit of ESHMP, therefore, is that it delivers scalability while adapting to a wide variety of workloads.

The remainder of this paper is organized as follows. Section 2 describes our motivation and reviews related work. Section 3 presents our scheduling algorithm, and Section 4 evaluates it. Section 5 concludes our work.

2. Related work

Several heterogeneity-aware schedulers were proposed in the literature [9,3,14,13]. According to our taxonomy described in Section 1, they employ either online monitoring or offline profiling.

Kumar et al. [9] and Becchi and Crowley [3] independently proposed two similar schedulers that adopt online monitoring approach. Both of them monitor performance of each thread on each core type and determine its performance improvement on fast core relative to slow core. They make thread-to-core assignment according to threads' performance improvement: those threads that have a higher fast-to-slow core IPC ratio gain a priority to run on fast core. Though these algorithms demonstrated improvements in performance on heterogeneity-agnostic ones, they have several shortcomings. As the number of core types on the chip increases, the sampling process becomes rather time-consuming, hurting their scalability vastly. Besides, threads are migrated between cores during sampling phases, which may cause load imbalance.

HASS [14] presented by Shelepov et al. uses offline profiling approach. It profiles application's architectural properties offline and embedded these properties into the application binary. By examining these properties, thread-to-core assignment is determined, and without any dynamic monitoring of applications' performance. HASS indeed avoids scalability barriers related to sampling and has a much simpler implementation, but it does not account for phase changes and is hard to adapt to various input sets, and thus it sacrifices accuracy. Moreover, it requires cooperation from developers to perform the steps needed for the generation of the architectural signature.

Balakrishnan et al. [2] designed a simple heterogeneity-aware scheduler for Linux, which makes sure that fast cores never go idle before slow cores. While this scheduler mitigates the effects of performance heterogeneity, it does not mean to improve efficiency. Mogul et al. [12] presented a scheduler that temporarily switches a thread to run on a slow core when the thread is executing a system call. By using system calls as a heuristic for thread assignment, this scheduler completely avoids any monitoring overhead (or the need to pre-generate architectural signatures), but it only applies to workloads dominated by system calls.

Li et al. [10] implemented a heterogeneity-aware algorithm in Linux, AMPS, which ensures that the load on each core is proportional to its capacity and that fast cores are never under-utilized. ESHMP also adopts the policy that faster cores are considered first before slower cores, but ESHMP makes such a assignment based on the efficiencies of threads in using fast cores, while AMPS does not, so AMPS may run a memory-intensive thread on a fast core and lower system throughput.

Teodorescu and Torrellas [16] developed an optimal algorithm for scheduling on *slightly* heterogeneous multicore architecture where core differences are caused by within-die process variation. By assuming that a thread's IPC is the same on all core types, a lot of overhead is avoided, though performance profiling is still required. The approach works well when cores are very similar to each other, but as compared to our scheduler, it is less applicable to highly heterogeneous systems.

3. ESHMP scheduler

In this section we describe the design of the ESHMP scheduler. We start with an overview of this scheduler, and then the details.

3.1. The scheduler

A performance heterogeneous multicore consists of cores with different power. For the threads running on these cores, their properties and resource demands may and often will be different. On such systems, performance improvement can only be realized when the scheduler map threads to cores in a way that will maximize the utilization of resources on each other.

In fact, the barriers to optimal scheduling on HMPs exist in obtaining the resource demands of threads, not in matching them to cores when their requirements are known. Moreover, the resource demands of threads are not constants but vary over time during different phases of its execution. Two approaches adopted in the existing work are offline profiling and online monitoring. Among them, offline profiling requires the coordination from the developers. To get an application's architectural signature, developers pre-execute the application with a typical input set and profile this execution process. But the selection of the typical input set is quite empirical and the properties generated by the typical input set cannot exactly stand for all other input sets' requirements. Furthermore, offline profiling is less easily applied to applications with a large number of phases, in that the amount of signatures embedded in the application binaries also grows as phase increases. Compared to offline profiling, online monitoring is more accurate and lightweight. Thus, we adopt the online approach.

The ESHMP scheduling algorithm has two steps: the first step fetches threads to cores and traces their execution, and the second step assigns these threads to cores according to threads' behavior and cores' features. Algorithm 1 shows the pseudo code of the ESHMP scheduling algorithm. The algorithm assumes that the given heterogeneous multicore processor consists of M cores, where they are indexed $1 \sim M$, and n threads, where they are indexed $1 \sim n$. The M cores are sorted in decreasing order of performance. The set of runnable threads is ordered so that the priority of t_i is greater than or equal to that of t_{i+1}.

The algorithm works as follows. If the number of the ready tasks is smaller than the number of the cores, each thread in the set is fetched and assigned to the core with the same index as the thread. Otherwise, only the first M threads that have higher priorities are fetched, and traced in the same way as used in the previous case. After sampling these threads on their corresponding cores, their efficiencies in using fast cores are obtained (measured by ASTPI, described in Section 3.2). Then, these monitored threads are reordered so that the thread with lower index has smaller ASTPI (i.e., greater efficiency in using fast cores). At last, these threads are sequentially assigned to the cores indexed from P_1 to P_r, where r is the number of the threads that have been monitored. Thus more CPU-intensive threads are assigned to faster cores while less CPU-intensive ones are assigned to slower cores. Note that this procedure is repeatedly performed in response to phase changes of applications. In this work, we adopt a moderate phase-detection mechanism in order to generate less migration as well as discover more phase changes. The selection of the period, however, is architecture dependent. In our implementation, we use 5 ticks for monitoring and 10 ticks for execution (roughly 1 tick $=$ 10 ms in our experimental system). That is, the period for our implementation is 15 ticks, among them 5 ticks are used to observe behavior and the rest are devoted to execution. Since our key objective is to evaluate the effectiveness of our method by comparing to the existing approaches, we do not pay much

Algorithm 1 The ESHMP scheduling algorithm

Assumption:
i indicates the index of the threads or the cores
r denotes the number of monitored treads
Inputs:
M cores $P_1, P_2, \ldots, P_M$, in decreasing order of performance
n threads $t_1, t_2, \ldots, t_n$, in decreasing order of priority

Begin
1: **Step 1: Measuring *ASTPI***
2: **if** $n < M$ **then**
3: $r \leftarrow n$
4: **else**
5: $r \leftarrow M$
6: **end if**
7: **for** $i = 0$ to r **do**
8: Measure *ASTPI* for thread t_i on core P_i
9: **end for**
10:
11: **Step 2: Assigning threads**
12: Sort the r threads in increasing order of *ASTPI* such that *ASTPI* of t_i is smaller than that of t_{i+1}
13: **for** $i = 0$ to r **do**
14: Assign thread t_i to core P_i
15: **end for**
End

attention to this aspect in this work. We are planning to consider more sophisticated phase-detection mechanisms in our future work.

As can be seen, ESHMP does not rely on the number of core types, thus outperforming existing online algorithms in complexity. Our evaluation results in Section 4 will confirm this point.

3.2. The new technique: ASTPI

In general, almost any thread gains speedup on fast cores relative to slow cores, but the degree of speedup may be different. To improve system throughput, however, among all available threads, only those threads with higher performance improvement on fast cores should be assigned to fast cores. As mentioned earlier, to measure speedup ratios, existing online monitoring approaches trace performance of threads on each core type; whereas monitoring ASTPI on any core is sufficient for ESHMP to discover efficiencies of threads in using fast cores.

The key insight is as follows. The completion time of a thread may be divided into two components: execution time, which is the amount of time that CPU spends in "real execution", and stall time, which is the amount of time that CPU spends in waiting for memory accesses. As mentioned before, for a thread, its stall time on each core type could be the same, since each core type has the same memory hierarchy (see Fig. 1). The ASTPI of a thread indicates the average time that the thread spends in waiting for memory accesses for executing one instruction. The smaller the ASTPI of a thread is, the greater the ratio of arithmetic operations to memory operations in the codes being executed by the thread, and the less the time it spent in waiting for memory accesses, thus the more the benefits it can gain on fast core type. In other words, the speedup of a thread on fast core relative to slow core is monotonically decreasing in the stall time of the thread. Since ASTPI of a thread implies the thread's efficiency in using a fast core, it is adopted as the technique for our scheduler.

First we demonstrate how ASTPI can be determined. Due to the completion time for a thread is composed of execution time and stall time, the stall time of the thread on core A can be determined by: $ST_A = CT_A - ET_A$, where ST_A, CT_A and ET_A are the thread's stall time, completion time and execution time achieved on core A, respectively. In addition, $CT_A = N_{instr}/IPS_A$, in which N_{instr} is the number of instructions of that thread, and IPS_A is the actual execution rate that the thread achieved on core A. Further, $ET_A = N_{instr}/MIPS_A$, where N_{instr} is the number of instructions of that thread, and $MIPS_A$ is the maximum IPS that core A is able to provide, which is achieved when no memory access happens. By substituting CT_A and ET_A with N_{instr}/IPS_A and $N_{instr}/MIPS_A$ separately, we have:

$$ST = \frac{N_{instr}}{IPS_A} - \frac{N_{instr}}{MIPS_A} = \frac{N_{instr} \times (MIPS_A - IPS_A)}{IPS_A \times MIPS_A}. \quad (1)$$

Moreover, ASTPI can be obtained by scaling down ST by N_{instr} times:

$$ASTPI = \frac{MIPS_A - IPS_A}{IPS_A \times MIPS_A}. \quad (2)$$

Then the definition for ASTPI can be formally given as follows:

Definition 1. ASTPI (Average Stall Time Per Instruction) for a thread is the average stall time that the thread takes for executing one instruction. It can be measured using the following formula:

$$ASTPI = \frac{MIPS_A - IPS_A}{IPS_A \times MIPS_A} \quad (3)$$

where $MIPS_A$ is the maximum IPS that core A can provide, which can be statically determined by the frequency of core A multiplied by the maximum IPC of core A (i.e., $MIPS_A = MIPC_A \times Frequency_A$), and IPS_A is the IPS that the thread achieved on core A, which can be dynamically monitored via hardware performance counters (i.e., $IPS_A = N_{instr}/CT_A$).[1]

At last let us explain the basic idea in detail. The speedup factor (SF) for a thread is the degree of performance improvement of the thread running on fast core relative to slow core, which is measured using the following formula:

$$SF = \frac{CT_{slow}}{CT_{fast}} \quad (4)$$

where CT_{fast} and CT_{slow} are the thread's completion times achieved on fast and slow cores respectively. Intuitively, for a thread, SF is the degree of the reduction of the completion time running on fast core relative to slow core. As the completion time for a thread is composed of execution time and stall time, the SF for the thread can be transformed into:

$$SF = \frac{ET_{slow} + ST_{slow}}{ET_{fast} + ST_{fast}} \quad (5)$$

where ET_{fast} and ET_{slow} denote the thread's execution times achieved on fast and slow cores individually, and ST_{fast} and ST_{slow} are the thread's stall times achieved on fast and slow cores

[1] A similar terminology has been defined in [6] but our definition does not conflict with it. The two terminologies are defined from different angles. The terminology proposed by Hennessy et al. [6] is defined using the sum of products of cache misses and related penalties of all levels. Since the cache hierarchy adopted in modern CPUs often has more than one level, it has more parameters (i.e., cache miss rates of all levels) than ours (i.e., IPS), incurring more complexity. More important is that modern CPUs only provide facilities for measuring the miss rate of the last level cache (LLC), which means it is impractical to measure average stall time per instruction (ASTPI) using just LLC. Thus a new way to determine ASTPI is presented in this work.

separately. Because ST_{fast} equals ST_{slow} (the stall times of a thread on all core types are equal), we have:

$$SF = \frac{ET_{slow} + ST}{ET_{fast} + ST}, \quad ST \geq 0 \tag{6}$$

where ST is the stall time which the thread achieved on any core. The ET of a thread on a core can be expressed as: $ET = N_{instr}/MIPS$, where N_{instr} is the number of instructions of that thread, and $MIPS$ is the maximum IPS that the core is able to provide, which is achieved when no memory access happens. Thus we can consider SF as a function of ST. Then the derivative of SF is as follows:

$$SF'(ST) = \frac{ET_{fast} - ET_{slow}}{(ET_{fast} + ST)^2}, \quad ST \geq 0 \tag{7}$$

Obviously, $SF'(ST)$ is negative since fast core takes less time to execute the thread than slow core does. So we may safely draw the conclusion that the speedup of a thread on fast core relative to slow core is monotonically decreasing in the stall time of the thread. In other words, the smaller the stall time is, the greater the speedup factor is. Similar conclusion can be drawn for ASTPI as well, since ASTPI is the stall time divided by the number of instructions of that thread. That is, the smaller the ASTPI is, the less the time the thread spends in waiting for memory accesses. The maximum and the minimum of SF are $SF(0)$ and $SF(+\infty)$ respectively, as it is monotonically decreasing in ST and $ST \geq 0$. They are calculated as follows:

$$SF(0) = \frac{ET_{slow}}{ET_{fast}} = \frac{MIPS_{fast}}{MIPS_{slow}} \tag{8}$$

where $MIPS_{fast}$ and $MIPS_{slow}$ denote the MIPSes that fast and slow cores are able to provide respectively.

$$SF(+\infty) = \lim_{ST \to \infty} SF(ST) = 1. \tag{9}$$

From above, we can see that the issue of determining threads' speedup is able to be converted into the issue of measuring threads' ASTPI. Therefor, we are able to use ASTPI as a technique to measure threads' efficiencies in using fast cores.

The experiment results in Section 4.2 show that the ASTPIs of a thread on all core types are the same and demonstrate that the technique of ASTPI closely approximates threads' efficiencies in using fast cores.

4. Evaluation

This section discusses our evaluation methodology and results.

4.1. Methodology

Two computers were used to evaluate the ESHMP scheduler. One was a dual-socket × quad-core Intel Xeon Clovertown server in which every quad-core package consists of two dual-core chips. Each core includes a 64 KB L1 cache, and each chip (two cores) has a shared 6 MB L2 cache. All cores run at frequencies of 2 and 2.83 GHz. A pair of cores on a package are within a voltage/frequency scaling domain, i.e., every dual-core chip is a frequency scaling unit. This one was mainly used to validate the accuracy of our method for estimating the efficiency of a thread in using a fast core, for it has only two frequencies available for scaling. Another was an AMD Operon system with two quad-core chips. Each core has private 64 KB instruction and data caches, and a private L2 cache of 512 KB. A 2 MB L3 cache is shared by the four cores on a chip. Each core is capable of running at a range of frequencies from 1.15 to 2.3 GHz. The frequency of each core is able to be varied independently, for each core is within its own voltage/frequency domain.

First, we created two configurations to test the accuracy of our method for estimating the thread's efficiency in using a fast core. Then, in order to validate that the ESHMP scheduler has an edge in scalability over the existing online approaches (such as IPC-Driven), we created three test configurations, and different configurations have different number of core types. For our algorithm is heterogeneity-aware, among those threads scheduled by it, memory-intensive threads are allocated to slow cores and CPU-intensive threads are allocated to fast ones. Thus, those threads on slow cores are more cache-greedy than those on fast cores. With such a scheduler, sharing a LLC among slow cores causes cache conflicts, while sharing a LLC among fast cores causes cache to be idle. We asserted that among HMP systems in which heterogeneity-aware schedulers are adopted and there are more than one LLC, the architecture where heterogeneous cores share LLCs gain better performance than the ones where homogeneous cores share LLCs, e.g., in Fig. 1 architecture (a) is better than architecture (b). As ESHMP, IPC-Driven and HASS are all heterogeneity-aware schedulers, we may expect that they all will perform better on systems configured with heterogeneous LLC-sharing mode. That is, they gain better performance on heterogeneous LLC-sharing setups because they are able to match threads to cores according to the characteristics of threads and cores. Moreover, the most heterogeneity-aware scheduler is bound to gain the most on heterogeneous LLC-sharing mode, as it is able to make the most accurate assignment. So, at last, another two configurations were created to validate the assertion and to evaluate which scheduler is more heterogeneity-aware. We created those test configurations by disabling some of the cores and setting cores to run at different frequencies using DVFS. All these test configurations are summarized in Table 1.

The benchmarks used to evaluate our algorithm were from the SPEC CPU 2006 suite. To simulate real workloads as far as possible, nineteen benchmarks with a wide variety of behaviors were selected from 29 ones of the suite, including memory-intensive (MI) ones (e.g., milc and soplex), CPU-intensive (CI) ones (e.g., namd and calculix) and multi-phased ones (e.g., astar and leslie3d). We constructed ten workloads with these selected benchmarks (see Table 2). These workloads can be classified into three categories: single-phased (SP), multi-phased (MP) and mix-phased. The benchmarks in single-phased workloads are single-phased, either memory-intensive or CPU-intensive. While the benchmarks in multi-phased workloads exhibit different phases across their execution (e.g., h264ref is a CPU-intensive application that also exhibits some memory-intensive phases), and mix-phased workloads consist of both single-phased and multi-phased benchmarks. The first five workloads are single-phased ones that combine homogeneous applications (4CI and 4MI) or mix CPU-intensive and memory-intensive applications (3CI-1MI, 2CI-2MI and 1CI-3MI). The workloads 4MP_A and 4MP_B are multi-phased ones. The remaining workloads are mix-phased ones. The workload names in the left column are listed in the same order as the corresponding benchmarks in the right column, so for example in the 3MP-1SP workload bwaves, leslie3d and astar are multi-phased (MP) applications and namd is a single-phased (SP) application.

For a test of a given workload, we launch its predetermined benchmarks, and when a benchmark terminates it is restarted repeatedly by a script until the longest benchmark in the workload completes three times. The average completion time of each benchmark under ESHMP is measured and compared to those under IPC-Driven and HASS. Since the ESHMP scheduler is proposed to make an improvement on IPC-Driven and HASS, we implemented all three of them on Linux 2.6.21 to validate this. In order to fairly compare our approach with IPC-Driven, for the implementation of IPC-Driven, we adopted the same phase-monitoring duration as ESHMP (i.e., 5 ticks, see Section 3.1).

Table 1
Test configurations.

Name	Core types / cores	Other information
Intel-2, 2	Core type.1:(2@2.0 GHZ) Core type.2:(2@2.83 GHZ)	For evaluating the accuracy of ASTPI
AMD-2, 2	Core type.1:(2@1.15 GHZ) Core type.2:(2@2.3 GHZ)	For evaluating the accuracy of ASTPI
AMD-2, 2	Core type.1:(2@1.4 GHZ) Core type.2:(2@2.3 GHZ)	Two core types
AMD-2, 1, 1	Core type.1:(2@1.4 GHZ) Core type.2:(1@2.0 GHZ) Core type.3:(1@2.3 GHZ)	Three core types
AMD-1, 1, 1, 1	Core type.1:(1@1.15 GHZ) Core type.2:(1@1.4 GHZ) Core type.3:(1@2.0 GHZ) Core type.4:(1@2.3 GHZ)	Four core types
AMD-2, 2	Core type.1:(2@1.4 GHZ) Core type.2:(2@2.3 GHZ)	Every two homogeneous cores share one L3 cache
AMD-2, 2	Core type.1:(2@1.4 GHZ) Core type.2:(2@2.3 GHZ)	Every two heterogeneous cores share one L3 cache

Table 2
Workloads.

Workload name	Benchmarks
4CI	perlbench, povray, sjeng, gromacs
3CI-1MI	calculix, povray, hmmer, mcf
2CI-2MI	namd, sjeng, soplex, omnetpp
1CI-3MI	gamess, GemsFDTD, milc, mcf
4MI	GemsFDTD, soplex, milc, omnetpp
4MP_A	h264ref, libquantum, leslie3d, astar
4MP_B	h264ref, dealII, bwaves, astar
3MP-1SP	bwaves, leslie3d, astar, namd
2MP-2SP	namd, gamess, leslie3d, astar
1MP-3SP	astar, sjeng, milc, mcf

That is, IPC-Driven monitors threads' execution for 5 ticks on each core type before making thread-to-core assignment.

The rest of the evaluation section is structured as follows. In Section 4.2 we evaluate the accuracy of our method for estimating a thread's efficiency in using a fast core. In Section 4.3 we explore the scalability of our approach associated with the number of core types. In Section 4.4 we analyze the efficiency of our approach in adapting to applications with various phases and behaviors. In Section 4.5 we discuss the influence of LLC-sharing mode on the performance of HMPs and some results are presented.

4.2. Accuracy of ASTPI

For all applications in SPEC CPU 2006 suite, we first compare their ASTPIs on all core types with each other to further confirm that the stall times of a thread on all core types are the same, and then we compare the estimated *SF* to the actual *SF* to evaluate the accuracy of ASTPI estimation. The variance of ASTPIs across different core types is collected by running the benchmark on all core types, determining ASTPIs on these cores, and calculating the ratio of the standard deviation (SD) of the ASTPIs to the average value of the ASTPIs. Actual *SF* is measured by running the application on the slow core, then on the fast core, and computing the speedup. Estimated *SF* is obtained from the ASTPI throughout the entire run of the application on only one core.

As shown in Fig. 2, the standard deviations of all applications from their average values are no more than 4%, so we may consider the variances of ASTPIs across core types for all applications are the same. Fig. 3 demonstrates that the estimates are accurate for all benchmarks on both platforms.

4.3. Scalability analysis

Scalability was one of the main focuses in ESHMP's design, and so various configurations with different number of core types were considered for evaluation. In this section, we will analyze how the speedup of ESHMP over the other two schedulers varies as setup complexity grows.

Figs. 4 and 5 compare the speedups of ESHMP over IPC-Driven for different number of core types with various workloads on the AMD platform. For both single-phased workloads (4CI, 3CI-1MI, 2CI-2MI, 1CI-3MI and 4MI) and non-single-phased workloads (4MP_A, 4MP_B, 3MP-1SP, 2MP-2SP and 1MP-3SP), the speedup of ESHMP over IPC-Driven grows drastically as the number of core types increases. That is because our method is constant in the number of core types while IPC-Driven is not, though they both adopt online monitoring approach and use the same period for monitoring. The speedup of ESHMP is not more than 11% when there are only two core types. The speedup grows more than 13% for the case that the number of core types increases from two to three, but within 11% for the case that the number of core types increases from three to four. The result is out of our expectation in that the speedup of ESHMP over IPC-Driven is not linear in the size of core types. We expected that the speedup growth could be the same for the above two cases since IPC-Driven traces on all core types and ESHMP monitors on only one core type. This result is due to IPC-Driven leads to load imbalance while monitoring on configuration AMD 2,1,1 (see Table 1), where all core types are not configured with the same number of cores thus cores of small core type (i.e., have less cores) have more threads running on them for monitoring purpose.

Results for the speedups of ESHMP over HASS for different number of core types are shown in Figs. 6 and 7. As expected, the speedups of ESHMP over HASS are similar for configurations with different number of core types (variances among them are less than 2%), in that it, likes our approach, dose not rely on the number of core types.

4.4. Efficiency analysis

Another emphasis in ESHMP's design was the ability to adapt to programs with various phases and behaviors, and thus applications with different behaviors and many phases were used for evaluation. In this section, we will discuss how the speedup of ESHMP over the other two schedulers alters as workload changes.

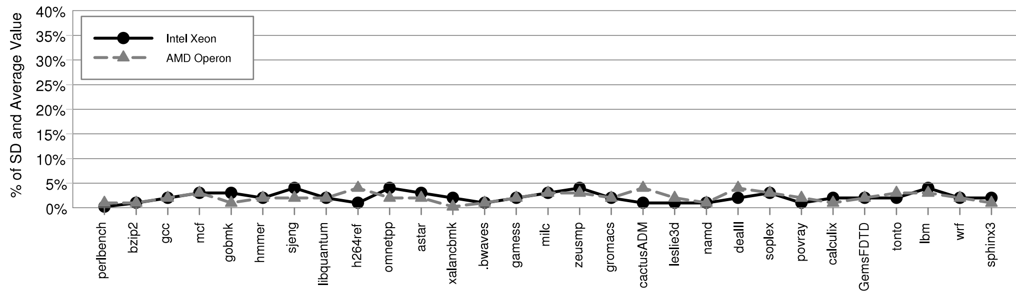


Fig. 2. Variances of ASTPIs across cores for all applications in SPEC CPU 2006 suite on an Intel Xeon and an AMD Opteron platforms.

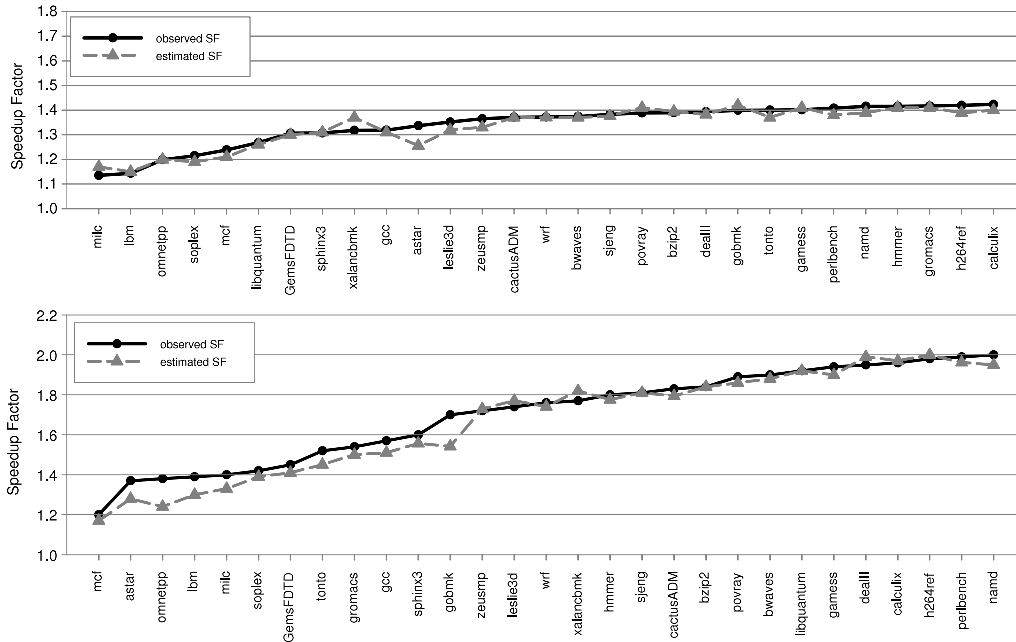


Fig. 3. Observed and predicted speedup factors for all applications of SPEC CPU 2006 suite on an Intel Xeon (top) and an AMD Opteron (bottom) platforms.

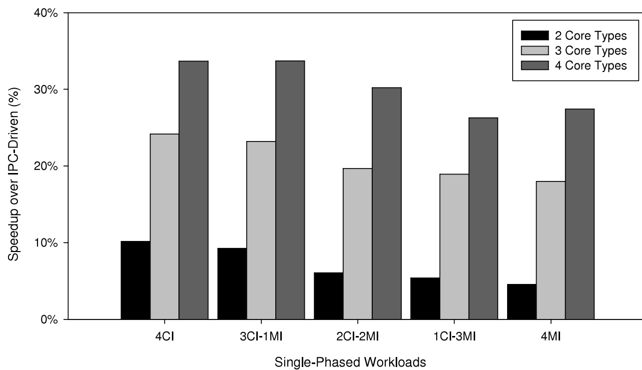


Fig. 4. Speedup of ESHMP over IPC-Driven for different # of core types with single-phased workloads on the AMD platform.

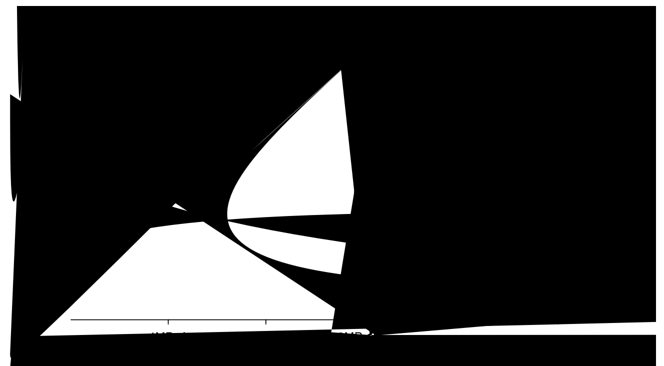


Fig. 5. Speedup of ESHMP over IPC-Driven for different # of core types with multi-phased and mix-phased workloads on the AMD platform.

In Fig. 6, we could observe that ESHMP even has a bit more overhead than HASS for single-phased workloads. That is because that our approach has to do an extra monitoring step to trace phase changes of applications during scheduling, but rare phase changes occur in single-phased workloads. Nevertheless, the overhead is so slight (no more than 2%) that it is ignorable. Besides, pure single-phased applications rarely exist in real workloads. In addition, for single-phased workloads including CPU-intensive applications, ESHMP is almost comparable to HASS (within 1%). That is because memory-intensive applications gain less speedup on fast cores than CPU-intensive ones do. In other words, if there are more CPU-intensive applications in the workloads, ESHMP would use more benefits from heterogeneity-aware scheduling to make up for overheads resulted from runtime monitoring. For workloads containing multi-phased applications (Fig. 7), HASS does not guarantee optimal scheduling and is worse than ESHMP by 5%–19%. The more multi-phased applications there are, the greater the speedup of ESHMP over HASS. For instance, the speedup for workload 4MP_A, which includes four multi-phased applications, is greater than that for workload 2MP-2SP, which only contains two multi-phased workloads. Overall, ESHMP outperforms HASS

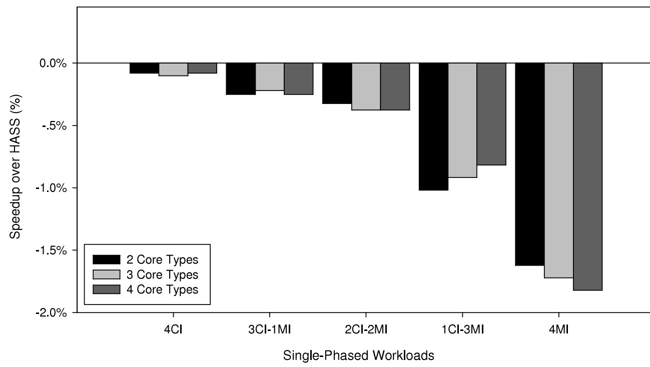


Fig. 6. Speedup of ESHMP over HASS for different # of core types with single-phased workloads on the AMD platform.

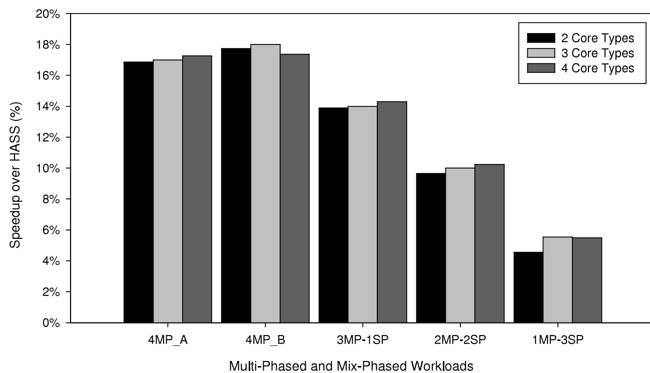


Fig. 7. Speedup of ESHMP over HASS for different # of core types with multi-phased and mix-phased workloads on the AMD platform.

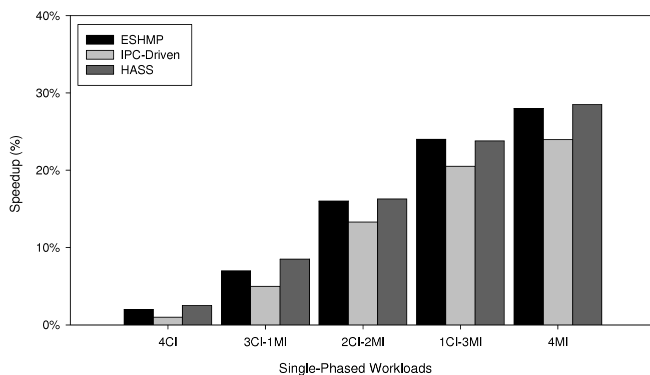


Fig. 8. Speedup of heterogeneous LLC-sharing mode over homogeneous LLC-sharing mode for different schedulers with single-phased workloads on the AMD platform.

vastly. The reason for the result is that HASS does not consider phase changes of applications.

Results for the speedups of ESHMP over IPC-Driven for different workloads are shown in Figs. 4 and 5. For the same reason as the previous case, ESHMP performs better for CPU-intensive applications as compared to IPC-Driven. For other workloads, ESHMP varies slightly (within 4%) as workload changes. That is because both ESHMP and IPC-Driven explore the same phase-detection mechanism for monitoring phase changes for all kinds of workloads.

4.5. Architecture analysis

Figs. 8 and 9 show speedups of ESHMP, HASS and IPC-Driven on the evaluation platform with heterogeneous LLC-sharing mode

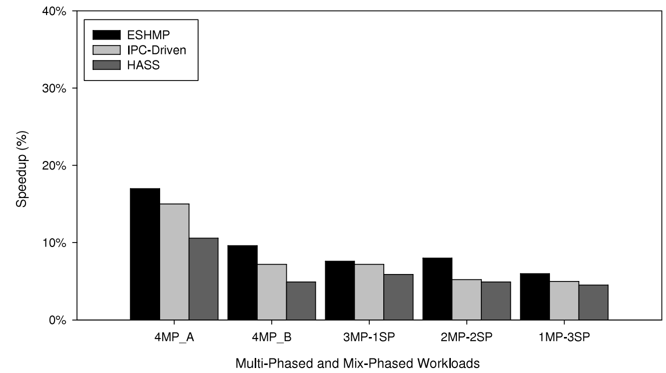


Fig. 9. Speedup of heterogeneous LLC-sharing mode over homogeneous LLC-sharing mode for different schedulers with multi-phased and mix-phased workloads on the AMD platform.

relative to that with homogeneous LLC-sharing mode. As shown in Fig. 8, for the workload that merely consists of CPU-intensive applications, the speedups for all schedulers increase no more than 3%. That is because CPU-intensive benchmarks are not as cache-greedy as memory-intensive ones, the change of LLC-sharing mode only has a weak influence on system throughput. For the workload consisting exclusively of memory-intensive applications, the speedups for the three schedulers are more than 23%. All the three heterogeneous-ware schedulers gain benefits from heterogeneous LLC-sharing mode, which confirms that our estimation is correct. IPC-Driven and HASS are not comparable, each of them outperforms the other for some kinds of workloads. As compared to IPC-Driven, ESHMP is definitely more efficient. Even though ESHMP has a bit more overhead (less than 2%) than HASS for single-phased workloads, it is more heterogeneity-aware on the whole, which is just the same as we analyzed in the previous subsection. At last, the fact that the relative speedups of ESHMP over other schedulers in Figs. 8 and 9 are not as large as that in Figs. 4–7 manifests that LLC-sharing mode has less influence on system performance than scheduling algorithm does.

5. Conclusions

We have proposed a new technique for measuring threads' efficiencies in utilizing fast cores, called ASTPI. We have shown that ASTPIs of a thread on all core types are the same and proved that the speedup factor of a thread is monotonically decreasing in ASTPI. Furthermore, we have provided an efficient and scalable scheduler based on the technique. Though a simple phase-detection mechanism was employed, through our evaluation of a real OS implementation on real hardware we determined that the ESHMP scheduler delivers scalability while adapting to a wide variety of applications. Besides, we discovered that heterogeneous-LLC sharing mode is better than homogeneous-LLC sharing mode for HMP systems with more than one cache, which could be used as a suggestion to hardware designers. The evaluation results on such settings also demonstrated that ESHMP is more heterogeneity-aware.

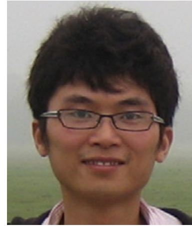
In this work, much more attention has been paid to performance estimation techniques, while less attention has been paid to scheduling mechanisms, in particular, phase-detection mechanism. Therefore, in the future work, we will focus on these aspects.

Acknowledgments

We are grateful to the anonymous reviewers for their valuable comments and suggestions. We believe that their suggestions will serve us well throughout our career. We would also like to express our gratitude to Xiaofeng Liu for giving us access to their servers.

References

[1] K. Asanovic, et al., A view of the parallel computing landscape, Communications of the ACM 52 (2009) 56–67.
[2] S. Balakrishnan, R. Rajwar, M. Upton, K. Lai, The impact of performance asymmetry in emerging multicore architectures, in: Proceedings of the 32nd International Symposium on Computer Architecture (ISCA05), 2005.
[3] M. Becchi, P. Crowley, Dynamic thread assignment on heterogeneous multiprocessor architectures, in: Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05), 2005.
[4] S. Borkar, Thousand core chips—a technology perspective, in: Proceedings of the 44th annual Design Automation Conference, 2007.
[5] L. Hammond, B. Nayfeh, K. Olukotun, A single-chip multiprocessor, Computer 30 (1997) 79–85.
[6] J.L. Hennessy, D.A. Patterson, A.C. Arpaci-Dusseau, Computer Architecture: A Quantitative Approach, vol. 1, Morgan Kaufmann, 2007.
[7] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, D.M. Tullsen, Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction, in: Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36'03), 2003.
[8] R. Kumar, D.M. Tullsen, N.P. Jouppi, P. Ranganathan, Heterogeneous chip multiprocessors, Computer (2005) 32–38.
[9] R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi, K.I. Farkas, Single-ISA heterogeneous multi-core architectures for multithreaded workload performance, in: Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04), 2004.
[10] T. Li, D. Baumberger, D.A. Koufaty, S. Hahn, Efficient operating system scheduling for performance-asymmetric multi-core architectures, in: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC'07), 2007.
[11] J.D. Meindl, Gigascale integration: is the sky the limit? IEEE Circuits and Devices Magazine 12 (1996) 19–24.
[12] J.C. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, V. Talwar, Using asymmetric single-ISA CMPs to save energy on operating systems, IEEE Micro 52 (2008) 56–67.
[13] D. Shelepov, A. Fedorova, Scheduling on heterogeneous multicore processors using architectural signatures, in: Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with the 35th International Symposium on Computer Architecture (WIOSCA'08), 2008.
[14] D. Shelepov, J.C. Saez, S. Jeffery, A. Fedorova, N. Perez, Z.F. Huang, S. Blagodurov, V. Kumar, HASS: a scheduler for heterogeneous multicore systems, Operating Systems Review 43 (2009) 66–75.
[15] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, B. Calder, Discovering and exploiting program phases, IEEE Micro 23 (2003) 84–93.
[16] R. Teodorescu, J. Torrellas, Variation-aware application scheduling and power management for chip multiprocessors, in: Proceedings of the 35th International Symposium on Computer Architecture (ISCA'08), 2008.



Pengcheng Nie received his B.Sc. degree from the School of Software at the Northwest University of China in 2005. Currently, he is working toward the Ph.D. degree at the Xidian University, China. His research interests include modeling and scheduling for parallel and distributed computing systems, parallel algorithms and system architectures. He is a member of the China Computer Federation, a member of the IEEE, the IEEE Computer Society, and a member of the ACM.

Zhenhua Duan is a professor of Computer Science at the Xidian University, Xian China. He obtained his B.Sc. and M.Sc. degrees from the Northwest University of China in 1982 and 1987, and the Ph.D. degree from the University of Newcastle upon Tyne in 1996. He worked as a research associate in three universities including the University of Ulster, the University of Newcastle upon Tyne and the University of Sheffield. His research interests concentrate on concurrent, real-time, and hybrid systems, including modeling, simulation, and verification of such systems. In addition, he is interested in temporal logic programming, formal languages and automata, and formal semantics. He is also interested in multicore programming. He is a senior member of the China Computer Federation, a senior member of the IEEE, the IEEE Computer Society, and a senior member of the ACM.