Predictive Heterogeneity-Aware Application Scheduling for Chip Multiprocessors

Jian Chen, *Member*, *IEEE*, Arun Arvind Nair, *Member*, *IEEE*, and Lizy K. John, *Fellow*, *IEEE*

Abstract—Single-ISA heterogeneous chip multiprocessor (CMP) is not only an attractive design paradigm but also is expected to occur as a consequence of manufacturing imperfections, such as process variation and permanent faults. Process variation could cause cores to have different maximum frequencies; whereas permanent faults could cause losses of functional units and/or cache banks randomly distributed on cores, resulting in fine-grained heterogeneous CMPs. Hence, application schedulers for CMPs need to be aware of such heterogeneity to avoid pathological scheduling decisions. However, existing heterogeneity-aware scheduling schemes rely on either trial runs or offline profiled information to schedule the applications, which incur significant performance degradation and are impractical to implement. This paper presents a dynamic and predictive application scheduler for single-ISA heterogeneous CMPs. It uses a set of hardware-efficient online profilers and an analytical performance model to simultaneously predict the applications performance on different cores. Based on the predicted performance, the scheduler identifies and enforces near-optimal application assignment for each scheduling interval without any trial runs or offline profiling. We demonstrate that, using only a few kilo-bytes of extra hardware, the proposed heterogeneity-aware scheduler improves the system throughput by an average of 20.8 percent and the weighted speedup by 11.3 percent compared with the commodity OpenSolaris scheduler. Compared with the best known research scheduler, the proposed scheduler also improves the throughput by 11.4 percent and the weighted speedup by 6.8 percent.

Index Terms—Heterogeneous multicore processor, application scheduling, performance modeling, process variation

1 Introduction

As transistor density and die size continue to grow, chip multiprocessors (CMPs) become increasingly susceptible to process variation and permanent faults caused by the inability to precisely control the manufacturing process. Process variation could result in large variation on transistor threshold voltage, which causes maximum operating frequencies to be different for the cores on the same die [1]. On the other hand, permanent faults would render parts of the core unusable, which may result in expensive yield loss. To guard against yield loss, it has been suggested to exploit the redundancy in the cores, and salvage them by disabling the faulty yet noncritical units, such as redundant functional units and cache SRAM arrays [2], [3]. This results in a relatively fine-grained single-ISA heterogeneous CMPs, in which each core has the same ISA but different frequency and/or different cache size, number of functional units, and so on.

Besides the heterogeneity caused by manufacturing, the single-ISA heterogeneous CMP itself has been demonstrated to be an attractive design alternative to its homogeneous counterpart [4]. By integrating cores with same ISA but different complexity in a single die, single-ISA heterogeneous CMP could significantly improve the execution efficiency for various workloads as it provides the hardware substrate to match the different workload requirements [4]. This design-caused CMP heterogeneity, combined with the manufacturing-caused CMP heterogeneity, underscores the importance of the single-ISA heterogeneous CMP, and presents a unique challenge to application scheduling in the operating system (OS).

Existing commodity application schedulers, such as the one in Linux or OpenSolaris [5], assume cores are symmetric, which could lead to pathological application scheduling decisions in the presence of core-level heterogeneity. For example, a memory-bound application with low instruction level parallelism (ILP) may be scheduled to a fast core; whereas a computation-bound high-ILP application may be scheduled to a slow core, resulting in poor overall performance. To prevent the undesirable scheduling results, researchers have proposed several heterogeneity-aware scheduling schemes by using trial runs or offline profiling. Kumar et al. [4] proposed a straightforward method that tentatively runs the program on different cores, each for a short period of time, and then schedules the program to the optimum core according to the sampled energy-delay product (EDP) during the tentative runs. Becchi and Crowley [6] extended Kumar's work by using the measured instruction-per-cycle (IPC) ratios between two different cores to determine the migration of the applications, which reduces the number of trial runs for each migration enforcement. The drawback of these methods is that the trial runs could incur significant power and performance overhead in moving around the architecture states and data sets, negating the benefit of the improvement in application scheduling. In addition, these

- *J. Chen is with Intel Corporation, Apt. 524, 2557 NW Overlook Drive, Hillsboro, OR 97124. E-mail: chenjian@utexas.edu.*
- *A.A. Nair is with the Adavanced Micro Devices, Apt. K302, 755 E Capitol Ave., Milpitas, CA 95035. E-mail: nair@utexas.edu.*
- *L.K. John is with the Department of Electrical Communication Engineering, The University of Texas at Austin, 1 University Station C0803, Austin, TX 78712. E-mail: ljohn@ece.utexas.edu.*

Manuscript received 25 Mar. 2012; revised 9 Aug. 2012; accepted 12 Aug. 2012; published online 28 Aug. 2012.
Recommended for acceptance by R. Gupta.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2012-03-0225.
Digital Object Identifier no. 10.1109/TC.2012.212.

methods also suffer from poor scalability because the number of trial runs required to explore all scheduling options grows almost *exponentially* as the number of heterogeneous cores increases. Shelepov et al. [7] proposed to utilize reuse distance signature constructed from offline profiling to schedule applications to cores with different cache sizes. Chen and John [8] used more comprehensive program characteristics profiled offline to find the proper program-core mapping. These offline techniques can only schedule the application statically, missing the opportunity to exploit program phase changes. Moreover, the need of offline profiling and encoding/decoding of the profiled information in the program binary could results in dramatic modification in the interface between OS, compiler and architecture, which fundamentally limits the applicability of the offline approach in practice.

To address these limitations, we present a new heterogeneity-aware application scheduler that *proactively* and *dynamically* schedules the applications to appropriate cores with the assistance of cost-effective online profilers. Unlike the existing dynamic schedulers, the proposed heterogeneity-aware scheduler leverages an analytic performance model to *simultaneously* predict the performance of an application on different cores, and formulates the scheduling decisions accordingly, without any trial runs. By collecting the application characteristics dynamically during the application's execution, the performance model can update the performance prediction at each scheduling interval, allowing the scheduler to dynamically adapt to program phase changes. In particular, the contributions of this paper are as follows:

- We build a comprehensive yet cost-effective online profiler, and an online analytic performance model that utilizes the online profile to accurately predict the performance of cores with different configurations on multiple resources. These resources include core frequency, L2 cache, and functional units, covering some of the most representative resources that are vulnerable to process variation and permanent faults. We show that the analytic model can predict the performance with an average error of 8.2 percent.
- We propose a framework for heterogeneity-aware application scheduling based on the proposed performance model. Our approach eliminates the need of trial-runs or offline profiling, yet can dynamically and efficiently adapt to program phases. We compare our approach with a set of dynamic scheduling schemes from prior work, and demonstrate that our approach improves throughput by an average of 20.8 percent as compared to the *OpenSolaris* scheduler, and by 11.4 percent as compared to the best known research scheduler.

This paper is organized as follows: Section 2 shows the background of process variation and permanent faults. Section 3 gives an overview of the proposed scheduling framework. Section 4 describes the performance model. Section 5 shows the structures of the online profilers. Section 6 presents the scheduling algorithms. Section 7 describes the experiment methodology. Section 8 evaluates the performance of the proposed scheduler. Section 9 describes the related works, and section 10 concludes this paper.

2 Background

Process variation is defined as a divergence in the parameters of the fabricated transistors from their nominal values, both within dies and die-to-die [1]. It occurs due to random dopant fluctuations and shortcomings of lithographic processes, and could significantly affect the threshold voltage of transistors. ITRS [9] reports that the 3σ intradie variation of a transistor's threshold voltage and effective channel length can be as large as 42 and 12 percent in 45-nm technology, and is expected to be worsen as the technology scales down further. The variation on these parameters directly impacts the switching speed of the transistors, which further causes the maximum operating frequency of the processor to deviate from its nominal value. In a multicore processor, this implies that different cores may have different peak operating frequencies.

Besides process variation, hard faults are another important issue in the manufacturing process. They are caused by imprecise calibration of the equipments, contaminants in the materials, as well as particle impurities in the air [2], and could incur functional failures in parts of the processors, resulting in expensive yield loss. It is expected that the yield loss will be exacerbated as the transistor density and die size increase, and needs to be carefully controlled. To mitigate yield loss, designers typically leverage the redundancy in processor components such as SRAM arrays, functional units and queues, to recover faulty processors by disabling some of the defective yet noncritical units [2]. These rescued processors are fully functional, albeit with reduced performance due to the reduction in certain hardware resources. That said, not all faulty units are suitable for this yield-enhancing technique: faults in critical units, such as control units, could cause complete failure of the processor; faults in reorder buffer (ROB) or load/store queue may require complex and expensive hardware to recover the functionality. Hence, in this paper, we focus on two types of representative resources that can be protected by this yield-enhancing technique, namely, available functional units (FU) and L2 cache size. Functional units have their natural redundancy in microprocessors, especially in wide-issue superscalar processors, and have been explored to improve the yield [2]. L2 cache occupies a large amount of chip real estate, and is susceptible to hard faults. These hard faults may be sporadically distributed across a few ways on different sets, or spatially correlated across multiple ways in the same set. In the former case, it has been shown that the defective ways in L2 cache can be discovered and disabled during manufacturing test, which results in a smaller, but functional cache [3]. In the latter case, a technique similar with horizontal yield aware power down (H-YAPD) [10] can be employed, which essentially remaps the addresses on the defective ways such that all ways of the same set are never disabled, and the cache behaves identical to a cache with fewer ways.

Both process variation and hardware faults are expected to coexist in the manufacturing process, and the compounded effect of these two has significant impact on CMP: each core in CMP may not only have different operating frequency but also different amount of functional units and L2 cache sizes, resulting in fine-grained

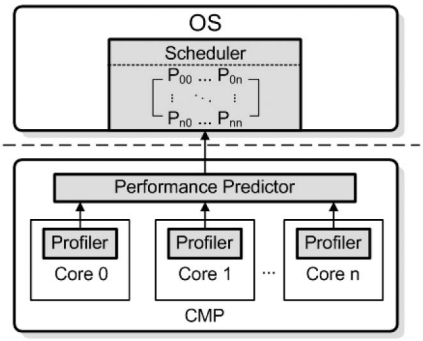

Fig. 1. The overview of the application scheduling framework.

heterogeneous CMPs. Moreover, such single-ISA heterogeneity may also be introduced into CMPs as an important design option to improve the power efficiency of CMPs. Therefore, there is an urgent need for a scheduling methodology that is aware of, and can efficiently exploit this heterogeneity, such that the overall throughput and efficiency of CMPs are optimized.

3 Overview of the Framework

The proposed framework for predictive heterogeneity-aware scheduling (PHASE) consists of three components: the online profiler, the performance predictor, and the scheduling heuristic, as shown in Fig. 1. The online profiler noninvasively profiles the application running on each core, and extracts the application's inherent characteristics required for the performance prediction. The performance predictor collects the profiled application characteristics at the end of each scheduling interval, and predicts the application's performance on other cores using the collected application statistics and the configurations of the corresponding cores. The predicted performance values are organized as a performance matrix and passed to the OS scheduler, where the scheduling algorithm identifies and enforces the appropriate assignment of the applications for the next interval based on the given objectives. As a result, the PHASE framework completely eliminates the need of trial runs, meanwhile it is able to dynamically and efficiently adapt to program phase changes.

Note that the proposed scheduling algorithm is not intended to replace, but rather complement the existing criteria for application scheduling. Specifically, the heterogeneity-aware scheduling is enforced only after the scheduler has chosen the applications from its application pool based on existing criteria including priority, fairness, and starvation-avoidance. Note also that while this framework can also be applied in single-ISA heterogeneous CMP caused by design, this paper focuses its application on heterogeneous CMPs resulting from process variations and manufacturing defects. In the following sections, we explain each component of the scheduling framework in detail.

4 Performance Predictor

Predicting/estimating the application's performance on different cores based on the observed application characteristics is the key step to avoid expensive trial runs. A naïve approach to do so is to use the measured IPC rate on one core as a proxy of the IPC rate on other cores [11], which might work when each core differs only in clock frequency. However, when each core has different number of functional units and/or different L2 cache sizes, this approach is no longer valid. In fact, an application may have very different performance on cores with the same clock frequency but different L2 caches or FU numbers. Therefore, in the presence of *fine-grain* heterogeneity on multiple resources, a comprehensive performance model is required to capture the impact of individual resource on the overall performance.

4.1 Basic Performance Model

The performance model is based on the previously proposed interval analysis [12], [13], which treats the exhibited cycle-per-instruction (CPI) rate as a sustained steady-state execution rate intermittently disrupted by long latency miss events, such as, L2 cache misses and branch mispredictions and so on. With the interval analysis, the total CPI of an application can be treated as the sum of three CPI components:

$$CPI_{total} = CPI_{exe} + CPI_{mem} + CPI_{other}, \quad (1)$$

where CPI_{exe} represents the steady-state CPI when the execution is free from any miss events. It is fundamentally constrained by the ILP of the application and the issue width of the processor. The ILP of the application is typically characterized by the critical dependence chain of the instructions in the *instruction window* (equivalent to *reorder buffer* in this paper). Assume an instruction window size w, and an average critical dependence chain length l_w. On an ideal machine with unit execution latency, l_w indicates the average number of cycles required to execute the instructions in the instruction window; hence, the average throughput is w/l_w. On a realistic machine with nonunit execution latency, this number should be further divided by the average execution latency lat_{avg} according to Little's law [12]. Therefore, the average ILP, μ_{avg}, can be obtained by $w/(lat_{avg} \cdot l_w)$, which also represents the steady-state execution rate if the instruction issue width is unlimited. However, for a realistic processor with limited issue width ν, the steady-state execution rate would be saturated at either the average ILP or the issue width, whichever is smaller. As a result, CPI_{exe} can be obtained by $1/min(\mu_{avg}, \nu)$.

CPI_{mem} represents the CPI penalty caused by the load misses in the last level cache (L2 cache in this paper). The total L2 load miss latency can be calculated by multiplying the number of L2 load misses N_{L2} with the average memory access latency lat_{mem}, assuming there are no multiple L2 load misses outstanding. In practice, to hide the load miss latency, L2 caches are usually nonblocking and multiple L2 load misses could be outstanding. Under this circumstance, it has been shown that the average load miss latency is reduced to lat_{mem}/m_{ovp} [12], where m_{ovp} is the average number of outstanding load misses. Therefore, CPI_{mem} can be calculated by $lat_{mem} \cdot N_{L2}/(m_{ovp} \cdot N_{inst})$.

CPI_{other} is the CPI component caused by other miss events, such as instruction cache misses, branch mispredictions and so on. In this paper, we assume that the resources related with these miss events remain the same across

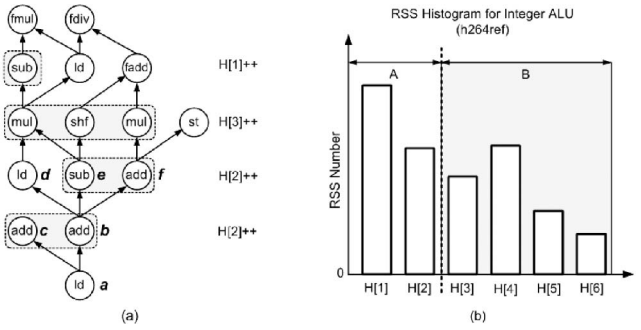

Fig. 2. The instruction ready set and the RSS histogram. (a) Example of an instruction dependence graph. (b) I-ALU RSS histogram for SPEC CPU2006 program *h264ref*.

different cores. Therefore, this CPI component can be treated as a constant parameter when a program is migrated from one core to anther as long as the program is in stable execution phase. Meanwhile, the value of this CPI component can be derived from the measurements on the core that the program is running on by transforming (1) to $CPI_{other} = CPI_{total} - CPI_{exe} - CPI_{mem}$, where CPI_{total} can be obtained from the performance counter, CPI_{exe} and CPI_{mem} can be derived from the observed program characteristics. Once CPI_{other} has been deduced, it can be plugged into the analytic model to estimate the performance of the program on other cores. As a result, we have our basic performance model as follows:

$$CPI_{total} = \frac{1}{min(\mu_{avg}, \nu)} + \frac{lat_{mem} \cdot N_{L2}}{m_{ovp} \cdot N_{inst}} + CPI_{other}.$$

4.2 Extended Performance Model

The basic performance model assumes that there are sufficient number of functional units (FU). However, when the number of FUs is limited, instructions may be stalled for additional cycles, which impacts the performance from two aspects. First, the additional stalled cycles increase the average execution latency, which in turn reduces the observed average ILP. Second, the limited number of functional units may also constrain the number of the instructions that can be issued in one cycle, causing the effective issue width ν_{eff} smaller than the nominal one.

To evaluate the performance impact of different FU numbers, we present the *ready set size histogram* for any given type of FUs. The *ready set* is the set of instructions in the instruction window that are ready for execution on a certain type of functional units, and the *ready set size* (RSS) is the number of instructions in the ready set, used as an index to the RSS histogram. Each time a new ready set is encountered, the histogram entry indexed with the corresponding RSS is incremented by one. As shown in Fig. 2a, when instruction a finishes execution, instruction b and c are ready to execute. Since both b and c will execute on integer ALU (I-ALU), the RSS for I-ALU is 2 and the corresponding entry in the I-ALU RSS histogram is incremented by 1. When instruction b finishes execution, instruction d, e, and f are free. Instruction d will execute on load unit. Both e and f will execute on I-ALU, though they have different opcodes. Hence, the new RSS for I-ALU is also 2. Note that even if at this point c is still in ready state, it should not be counted in the new ready set. Therefore, RSS histogram reflects the inherent property of the workload, and is independent of microarchitecture.

With the RSS histogram, we are able to estimate the number of stalled cycles and the effective issue width for any number of FUs. As shown in Fig. 2b, the number of I-ALU divides the histogram into region A and region B. The RSS in Region A is no larger than the I-ALU number; hence, instructions in this region do not experience additional stalled cycles caused by the limited number of I-ALU. While in region B, the I-ALU number is smaller than RSS, causing additional waiting cycles on the ready instructions. Assuming n fully pipelined I-ALUs and a ready set with RSS of m, it takes $\lfloor m/n \rfloor$ additional cycles to finish issuing the instructions in this ready set, contributing an additional cycle-instruction product $\sum_{i=1}^{\lfloor m/n \rfloor}(m - i \cdot n)$ to the equation of calculating the average instruction latency. Therefore, considering the additional stalled cycles caused by the limited number of FUs, the average instruction latency could change significantly, resulting in a modified observed average ILP, which we refer to as μ'_{avg}. On the other hand, instructions in region A and instructions in region B have different observed issue width. While the observed issue width for the instructions in region A equals the physical issue width, the observed issue width for those in region B is limited by the FU number n. Therefore, on average, the effective issue width $\nu_{eff} = pn + (1-p)\nu$, where p is the percentage of instructions in region B among the total instructions executed. As a result, with the limited functional units, CPI_{exe} becomes $1/min(\mu'_{avg}, \nu_{eff})$.

Besides modeling the impact of limited FU numbers, the basic performance model also needs to be augmented to capture the performance impact of different clock frequencies. This could be achieved by converting the CPI to the absolute execution time, which leads us to the following extended performance model:

$$\begin{aligned} Delay &= N_{inst} \cdot CPI_{total}/f \\ &= \frac{N_{inst}}{min(\mu'_{avg}, \nu_{eff}) \cdot f} + \frac{N_{L2} \cdot t_{mem}}{m_{ovp}} + C_{other}/f, \end{aligned} \quad (2)$$

where f is the core clock frequency, and t_{mem} is the absolute memory access time.

5 Online Profiler

The proposed performance model requires a set of program characteristics to derive the key parameters used in the model. These characteristics include: 1) the critical dependence chain, for deriving the average ILP; 2) the instruction ready set size histogram, for calculating the effective issue width with different FU number; and 3) the stack distance histogram [14], for estimating the number of L2 load misses with different L2 cache sizes. This section presents a set of noninvasive and cost-effective online profilers to dynamically extract these characteristics during the application's execution.

5.1 Critical Dependence Chain Profiler

The critical dependence chain (CDC) in this paper refers to the longest instruction dependence chain in the instruction

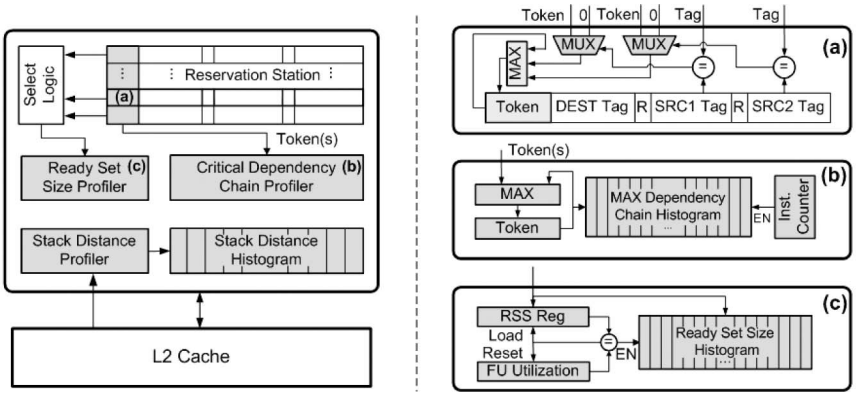

Fig. 3. The online profilers. (a)-(b) The critical dependence chain profiler. (c) The ready set size profiler.

window. To capture the CDC length, we propose a token-passing technique inspired by Fields et al.'s work [15]. A token is a field in each reservation station entry that keeps track of the dependence chain length, as shown in Fig. 3a. When an instruction enters the reservation station, its token field is set to zero; when an instruction leaves the reservation station for execution, its token field is incremented by one. The incremented token is propagated along with the result tag of the instruction. When the instruction finishes execution and its result tag matches the source tag of the waiting instruction, the propagated token is compared against the token of the waiting instruction, and the larger one is stored in the token field of the waiting instruction. Hence, by the time an instruction is ready for execution, its token holds the length of its longest dependence chain.

The CDC profiler compares the token of every issued instruction, and keeps track of the largest token, which is then used as an index to the *max dependence chain histogram*. As shown in Fig. 3b, the histogram is controlled by an instruction counter that monitors the number of issued instructions. When this number reaches the size of instruction window, the histogram entry indexed with the maximum observed token is incremented by 1, and the register that holds the maximum token is reset to zero. Consequently, the maximum dependence chain histogram holds the information of the longest dependence chain length for each instruction window. At the end of each scheduling interval, this histogram is used to calculated the average CDC length, and then reset to zeros for the next scheduling interval.

5.2 Ready Set Size Profiler

The ready set size profiler takes advantage of the standard instruction selection logic [16], where the information about the number of ready instructions on a certain type of FUs is readily available. This information is steered to the RSS register in the ready set size profiler for the corresponding FU type, as shown in Fig. 3c. Besides the RSS register, the ready set size profiler also contains a utilization counter that is incremented each time an instruction is issued to the corresponding FU for execution. When the utilization of the FU equals the previously stored RSS value, the RSS register is loaded with a new RSS value, and the utilization counter is reset to zero. Meanwhile, the RSS histogram entry indexed by the new RSS value is incremented by one. At the end of scheduling interval, RSS histogram is used to update the average instruction latency, and reset to zero.

Such profiling mechanism can precisely capture the RSS information assuming the instructions are issued in the *oldest first* order. For a different instruction selection policy, the profiled RSS histogram may not exactly reflect the application's RSS statistics. Nevertheless, we expect that the discrepancy is small and its impact on the accuracy of performance prediction is negligible.

5.3 Stack Distance Profiler

To estimate the number of L2 load misses for different cache sizes, we employ Mattson's stack distance model at the granularity of cache ways [14], [17]. This stack distance model exploits the inclusion property of least recently used (LRU) replacement policy, i.e., the content of an N-way cache line is a subset of the content of any cache line with associativity larger than N. As an example, Fig. 4 shows the stack distance histogram of program *xalancbmk* on an eight-way associative cache, organized from MRU position to LRU position. For caches with the associativity reduced to six-ways (dash line in the figure), the data with stack distance larger than 6 cannot be hold in the cache, generating cache misses. Therefore, with the stack distance histogram, we are able to estimate the cache miss rate for any cache ways less than the profiled ways and consequently derive the number of L2 misses. Note that although the cache miss rate can be collected with the standard performance counters, this miss rate cannot be used to estimate the miss rate on caches with different sizes.

Profiling the stack distance requires an auxiliary tag directory (ATD) and hit counters for each cache set [17]. The

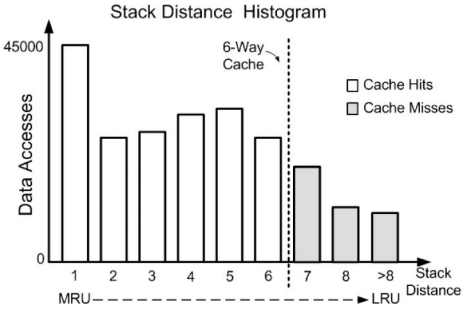

Fig. 4. Stack distance histogram of SPEC CPU2006 program *xalancbmk*.

TABLE 1
Hardware Cost of the Online Profilers

Profiler	Components	Costs(bit)
Critical Dependency Chain Profiler	token fields	7*128
	multiplexors, comparators	(7*2+7)*96
	histogram counters	32*128
RSS Profiler	RSS Reg & Utilization Reg.	7*2
	histogram counters	32*128
Stack Distance Profiler	LRU bits per ATD entry	3
	valid bits per ATD entry	1
	address bits per ATD entry	12
	ATD cost(128 sampled sets)	(3+1+12)*8*128
	Hit Counters	32*(8+1)
Cost of Profilers Per Core		27790

ATD has the same associativity as the largest L2 cache in the chip and keeps track of LRU replacement; and the hit counter counts the number of hits on each cache way. To reduce the hardware cost of ATD, we employ the dynamic set sampling (DSS) technique, which essentially uses a few sets to approximate the entire cache behavior [17]. In this paper, we sample 1 set out of every 32 sets.

5.4 Profiling for Other Parameters

Other parameters required by the analytic model can be obtained from the standard performance counters. For example, the performance counters in Intel Core architecture [18] are able to provide the instruction mix and cache hit/miss statistics. With these statistics, the average latency lat_{avg} can be derived by weight-averaging the percentage of each instruction type with the corresponding execution latency. Note that the load that misses L1 cache but hits in L2 cache is treated as an instruction with long execution latency. This average latency is further adjusted with the RSS histogram to count in the effect of limited functional units.

Similarly, the average memory level parallelism m_{ovp} can be obtained by monitoring the *miss status holding register* (MSHR) in L2 cache. Specifically, each time an L2 load miss occurs, we look up the MSHR for outstanding load misses. m_{ovp} is the average number of these outstanding misses across all L2 load misses.

5.5 Hardware Cost Analysis

The hardware cost of the profilers depends on the instruction window size as well as the L2 cache size. Assuming 128-entry instruction window, 96-entry reservation station, 32-bit physical address space, and 2-MB 8-way L2 cache with 64B block size, the total hardware cost amounts to 3.5 KB, as shown in Table 1. The hardware cost may be further reduced by using a smaller number of histogram counters based on the observation that most of the RSS or the CDC length is far smaller than the instruction window size. However, even without such optimization, the online profilers incur no larger than 0.2 percent hardware overhead on a core with 2-MB L2 cache. Note that these profilers are not in the critical path, and do not affect the processor's performance.

The computation cost of the performance prediction is mainly caused by converting the histograms to the parameters used in the performance model, which requires about 300 multiply-accumulate operations. In addition, the performance model itself needs 2 add, 1 comparison, 2 multiply, and 3 divide operations. Therefore, predicting the performance of an application on four cores requires around 350 arithmetic operations. Since the prediction is made only once every scheduling interval, these operations can be carried out by exploiting the FUs already on the chip. Specifically, the processor steers the data from the profilers to the idle FUs and uses their idle cycle slots to do the calculations needed in the performance prediction. These calculations are controlled by an embedded microcode sequence, which is invisible to the software and does not introduce additional regular instructions into the ROB. Since the FUs are usually underutilized due to the load/store instructions, by starting the performance prediction several thousands of cycles before the end of the scheduling interval, there should be sufficient idle slots to complete the necessary calculations. Hence, the computation cost of the performance prediction can be completely hidden behind the normal execution.

6 Scheduling Algorithms

With the online profilers and the performance predictor, the core-level heterogeneity of the CMP is exposed to the scheduler as a matrix of performance values (in the form of IPC or IPC speedup depending on the optimization target). The PHASE scheduler can simply searches through this matrix for the optimum program-core allocations, fundamentally eliminating the need of trial runs.

To identify the optimum program-core allocation from the performance matrix, a naïve approach needs exhaustive search, which has the complexity of $O(n!)$ and is not scalable. In contrast, our PHASE scheduler uses a greedy algorithm with polynomial computation complexity, as shown in Pseudocode 1. The algorithm first searches the estimated performance matrix for the largest entry, and the corresponding program and core indices are stored in the program-core allocation array and then removed from the index arrays. This process repeats for the remaining matrix until all indices of applications or cores have been visited. The newly obtained program-core allocation is enforced in the next scheduling interval only when the predicted performance gain is larger than the given *migration threshold*. This threshold serves as a *migration throttling* agent, which prevents applications from migrating when there is insufficient performance improvement to compensate the migration cost.

Pseudocode 1. Algorithm for Predictive Heterogeneous-Aware Scheduling.

```
#define N_c /*the number of cores in the CMP*/
#define N_p /*the number of programs to be scheduled*/
#define P_th /*the performance threshold*/
#define perf[N_p][N_c] /*the array of predicted performance*/
#define prog[N_p] /*the program index array*/
#define core[N_c] /*the core index array*/
#define core_alloc[N_c] /*the core allocation array*/
for (i = 0; i < N_c; i++)
    foreach n_c in core[N_c]
```

TABLE 2
Nominal Configurations of the CMP System

	Parameter	Configurations
Core	Clock Frequency	4GHz
	Fetch/Issue/Commit Width	4/4/4
	Ld/St Units	2/2
	I-ALU/FP Units/FP Multipliers	4/2/2 (fused multiply/add for I-ALU)
	ROB/Load Queue/Store Queue Size	128/32/32
	Branch Predictor	YAGS, 16 PHT bits, 10 Tag bits
Cache	L1 I-Cache/D-Cache	32KB, 2-way, 64B block, LRU, 1-cycle latency
	L2 Cache	2MB per core, 8-way, 64B block, LRU, 12-cycle latency, 32 MSHRs
	Coherence Protocol	Directory-based MOESI
Memory	Size/Model/Controller	4GB/DDR2-800/FR-FCFS policy[24]
	Organization	8 banks per rank, 2 ranks per DIMM

foreach n_p in $prog[N_p]$
if ($perf[n_p][n_c] > max\_perf$)
$N_{pmax} = n_p$; $N_{cmax} = n_c$;
$max\_perf = perf[n_p][n_c]$;
end if
end foreach
end foreach
$core\_alloc[N_{cmax}] = N_{pmax}$;
$total\_perf = total\_perf + max\_perf$;
remove N_{pmax} from $prog[N_p]$;
remove N_{cmax} from $core[N_c]$;
end for
if $((total\_perf - total\_mon\_perf)/total\_mon\_perf > P_{th})$
enforce schedule based on $core\_alloc[N_c]$;
end if

The complexity of this algorithm is $O(n^2 \cdot m)$, where n is the number of cores and m is the number of programs to schedule ($m \leq n$). Note that if the number of programs is larger than the number of cores, the scheduler will first choose the programs from the program pool based on the criteria such as priority and fairness, and then the performance matrix associated with these programs will be searched for the optimum allocation. Without losing generality, this paper focuses on the case that the number of programs is no larger than the number of cores.

Besides this proposed algorithm, we also evaluate a set of other scheduling algorithms for comparison. These algorithms include:

OpenSolaris: This is the default OpenSolaris scheduler, which is unaware of the core-level heterogeneity and treats each core as symmetric. It has the property of natural binding, that is, when an application gets scheduled to one core, this application is unlikely to be migrated to a different core in the next scheduling interval to avoid migration overhead [5]. Therefore, it can be treated as random static mapping, and is used as the baseline scheduler in this paper.

Becchi+: This algorithm is based on the one proposed by Becchi and Crowley [6]. While the original proposal only applies to two types of cores, we extend it to support four or more different cores. Specifically, the algorithm allows the applications run for one interval, and then it randomly selects two cores, swaps the applications running on the cores, and makes them run for another interval. The allocation that gives the higher performance between these two intervals is enforced in the next scheduling interval. To mitigate the overhead of program swapping, we allow the procedure to repeat 10 times and then no program swapping is allowed in the following 10 scheduling intervals, and then the swapping procedure repeats again for 10 times and so on.

Oracle: This algorithm assumes the performance of the program on different cores in the next scheduling interval is known a priori. It uses these *future* performance data to find the program-core allocation that gives highest throughput (or speedup), and enforce the allocation in the next scheduling interval. While it is unrealistic in practice, it sets an upper bound of the potential performance improvement.

Worst static scheduling (WSS): This is essentially the static program-core mapping that gives the lowest aggregated throughput (or speedup). It is only used as a reference point to highlight the worst situation that a heterogeneity-unaware scheduling scheme could possibly end up with.

7 Experiment Methodology

7.1 Simulation Platform

We use Simics [19], extended with the Gems toolset [20], to simulate a quad-core SPARCv9 CMP system running under OpenSolaris operating system. Each core in the CMP is four-issue out-of-order processor modeled by Opal [20]. The simulated CMP system contains a detailed memory subsystem model, which includes an intercore last-level cache network and a detailed memory controller. In addition, the simulated system supports software prefetching and next-line hardware prefetching. Table 2 lists the nominal configurations of the CMP system in detail. We use Wattch [21] to estimate the dynamic power, and Cacti 5 [22] to estimate the leakage power on caches and other SRAM structures in the core. We also use Orion [23] to estimate the power on the interconnection network of the last level cache. Therefore, the performance and power overhead of application migration is fully modeled in each application scheduling scenario.

This paper focuses on the core-level heterogeneity on clock frequency, integer ALU number and L2 cache size, yet it is infeasible to evaluate every possible configuration. Therefore, we evaluate three sets of heterogeneous configurations created by varying these resources over their nominal values, as shown in Table 3. These configuration sets are: low heterogeneity (LH) where only clock frequency varies, medium heterogeneity (MH) where both clock frequency and I-ALU number vary, and high heterogeneity (HH)

TABLE 3
Configurations of the CMP System

Parameter	Configurations											
	Low Heter.				Medium Heter.				High Heter.			
	C0	C1	C2	C3	C0	C1	C2	C3	C0	C1	C2	C3
Freq.(GHz)	4	3.6	3.2	2.8	4	3.6	3.2	2.8	4	3.6	3.2	2.8
I-ALU	4	4	4	4	4	2	3	1	4	2	3	1
L2 Cache (MB,Ways)	2, 8	2, 8	2, 8	2, 8	2, 8	2, 8	2, 8	2, 8	2, 8	1.5, 6	1, 4	0.5, 2

where all three resources vary. Note that not all resources are varying at the same direction. For example, while the clock frequency of C1 is larger than that of C2, the I-ALU number of C1 is less than that of C2. Although there are other heterogeneous configuration sets, these three configuration sets are representative in covering different degrees of heterogeneity. More importantly, these heterogeneity levels are used to demonstrate the superiority of PHASE over existing scheduling schemes regardless of the detailed heterogeneous configurations.

7.2 Workloads

To stress the scheduling in heterogeneous CMPs, the workload also needs to be heterogeneous (Homogeneous workloads, such as the threads spawned from a program, benefit little from scheduling in heterogeneous CMPs.) [7]. Therefore, we construct nine multiprogrammed workloads from the programs in SPEC CPU2006 benchmark suite [25], with each program compiled to SPARC ISA. Each heterogeneous workload contains two integer programs and two FP programs, as shown in Table 4. The program mix is based on the similarity analysis by Phansalkar et al. [26], and is created such that: 1) the workloads cover all representative programs; 2) programs in each workload are from clusters with large linkage distances [26]. Each workload is executed on the aforementioned three heterogeneous CMP systems. During the execution, each workload is fast-forwarded 3 billion instructions, and the next 100 million instructions are used to warmup the cache subsystem. We then simulate the full system for a time span equivalent to 1 second on a real 4-GHz CMP system, which covers up to 5 billion instructions for a program. The scheduling interval is set to 10 ms (standard in *OpenSolaris*). Therefore, each simulation gives us 100 scheduling intervals.

7.3 Metrics

We use the aggregated throughput, defined as the sum of each application's million-instructions per second (MIPS), and the weighted speedup [27], defined as $\sum_i IPC_i^{scheduled}/IPC_i^{ref}$, as the metrics to evaluate the system performance. In the weighted speedup, $IPC_i^{scheduled}$ is the IPC of the application i being scheduled with a certain scheduling algorithm, and IPC_i^{ref} refers to the IPC of the application i under the baseline scenario (OpenSolaris in this paper). To measure the efficiency of the system, we use the metric $mips^3/W$, which is inverse to energy-delay-square (ED^2) and has been accepted as the efficiency metric for high-performance systems [28].

TABLE 4
Workloads and Their Characteristics

Program Mix	Symbol	Category[1]
mcf,bwaves,povray,gcc	mbpg	M-M-C-C
xalancbmk,namd,lbm,omnetpp	xnlo	C-C-M-M
libquantum,xalancbmk,wrf, soplex	lxws	M-C-M-M
milc, soplex, omnetpp, sjeng	msos	M-M-C-C
leslie3d,sphinx3,hmmer,astar	lsha	M-C-C-C
zeusmp, libquantum, omnetpp, tonto	zlot	M-M-C-C
calculix, dealII, perlbench, bzip2	cdpb	C-C-C-C
povray, mcf, cactusADM, astar	pmca	C-M-M-C
milc, gobmk, lbm, gcc	mglg	M-C-M-C

8 Evaluation

8.1 Model Accuracy

The accuracy of the analytic performance model could largely impact the effectiveness of the proposed scheduling framework. To evaluate this accuracy, we run every SPEC CPU2006 program on a simulated processor for one scheduling interval, and use the performance model to estimate the program's CPI on other processors with different configurations. Meanwhile, we also run the programs on those processors for one scheduling interval and compare the observed CPI with the estimated one. As shown in Figs. 5a, 5b, and 5c, the average error between the estimated CPI and the observed one is no larger than 8.17 percent, indicating the performance model keeps a good track of the observed performance when only one resource varies its configuration. Fig. 5d shows the Monte Carlo simulation of 300 random configurations when all three resources vary simultaneously. The average error between the estimated CPI and the observed one is 6.71 percent. These errors are mainly due to: 1) the fact that the profiled critical dependence chain based on the number of dependent instructions may not be the *real* critical dependence chain in terms of execution latency; 2) the usage of dynamic set sampling to approximate the behavior of the entire cache; and 3) the fact that the simulator models hardware prefetching but the analytic model does not captures the effect of hardware prefetching. Nevertheless, we believe that these are the reasonable tradeoffs between the accuracy and the hardware cost, since the accuracy of the model is sufficient for our scheduling heuristic to achieve near-optimal performance.

8.2 Migration Threshold

As explained in Section 6, the proposed scheduling algorithm uses the migration threshold to control performance gain and throttle nonbeneficial program migration. The migration threshold should be reasonably high to filter out detrimental program migrations whose migration

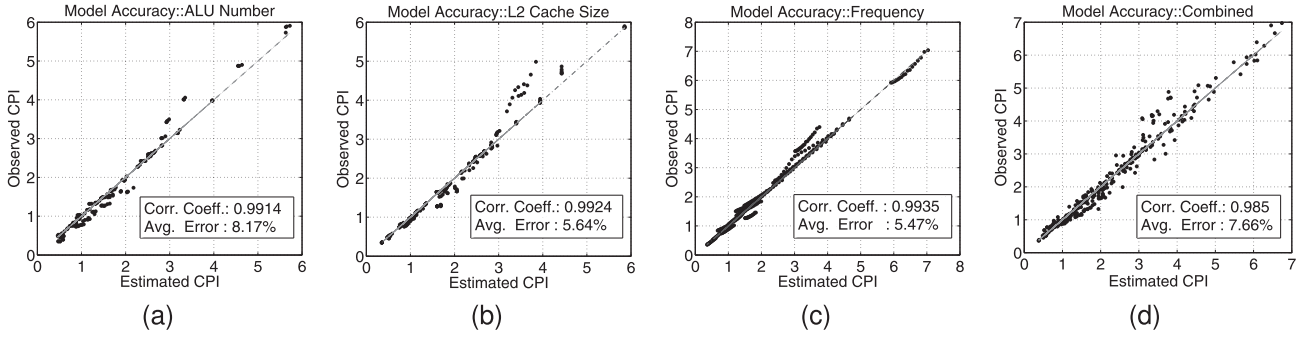

Fig. 5. Model accuracy. (a) The number of IALU varies from 1 to 4. (b) The L2 cache size varies from 512 KB to 2 MB at the step of 256 KB. (c) Frequency varies from 2 GHz to 4 GHz at the step of 0.1 GHz. (a)-(c) Only one resource changes with others in nominal configurations. (d) Three hundred random configurations when three resources vary simultaneously.

overhead is larger than the potential performance gain. However, the threshold cannot be too high, otherwise it may conservatively prevent the program from migrating, missing the opportunities for performance improvement. Therefore, a good threshold should prevent most of the detrimental program migrations yet still allow most of the beneficial ones. In this paper, we find out that 5 percent is a reasonable threshold value that meets this criterion for all workloads. One can further improve the performance by using an adaptive threshold, but it is beyond the scope of this paper.

8.3 Performance

A case study. As an example, Fig. 6 shows the details of program-core allocation for workload *lxws* under four different scheduling algorithms. As shown in Fig. 6b, *Becchi+* needs many scheduling intervals to try different program-core allocations before it stabilizes. After being stable for several intervals, *Becchi+* has to go back to the stage of trial runs again to detect any program phase changes that may cause changes in program scheduling, which significantly undermines the benefit of program scheduling. In contrast, PHASE completely avoids this problem by replacing the heavy-weight and slow trial runs with the light-weight and fast performance prediction. As shown in Fig. 6c, PHASE enforces a new program-core allocation immediately after the first scheduling interval. During the execution, PHASE also dynamically enforces different program-core allocation along with the overall performance changes. Note that while the performance of PHASE is close to Oracle, the program-core allocations in PHASE does not always match *Oracle*. This phenomenon mainly comes from two sources: 1) PHASE uses the history information to estimate future performance, hence cannot capture the sudden performance change in the next scheduling interval; whereas the *Oracle* scheduler knows the future events, and can adjust the scheduling decisions accordingly; and 2) due to the greedy nature of the searching algorithm, PHASE may be trapped in the application assignment that is only local optimum whereas the Oracle scheduler always enforces the global optimum assignment.

Improvement on throughput. Fig. 7a shows the comparison of the aggregated throughput for different scheduling policies. We observe that the performance of the *OpenSolaris* scheduler can be very close to (e.g., workload *mbpg*) or

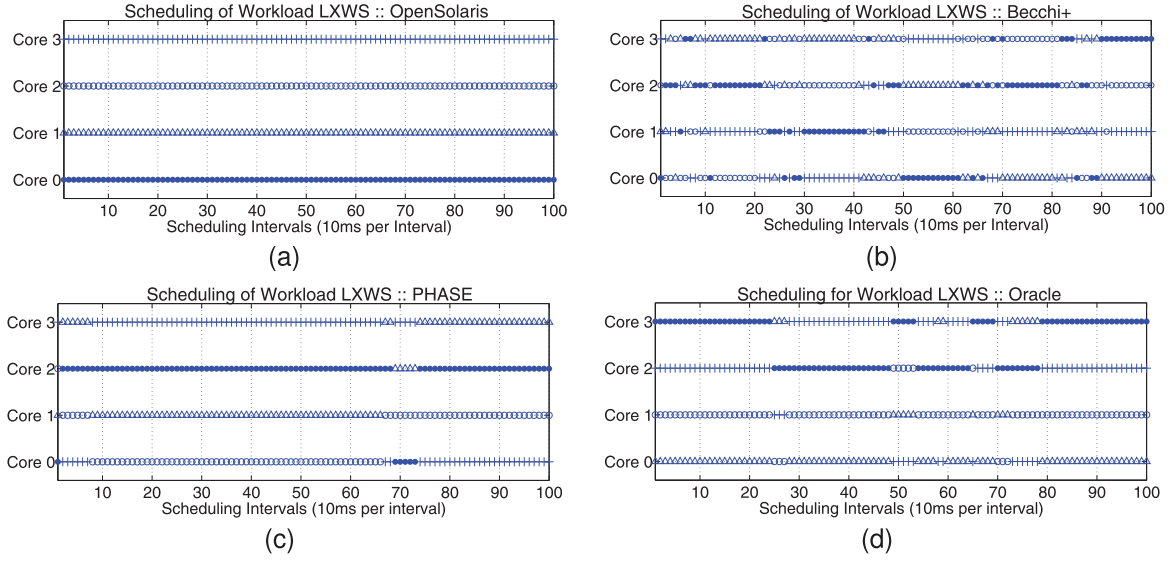

Fig. 6. Scheduling of the workload lxws. (a) OpenSolaris scheduling, and the normalized throughput is 1. (b) Becchi+ scheduling, and the normalized throughput is 1.030. (c) PHASE scheduling, and the normalized throughput is 1.084. (d) Oracle scheduling, and the normalized throughput is 1.102. For all subfigures, • stands for libquantum, △ for xalancbmk, ◯ for wrf, and + for soplex.

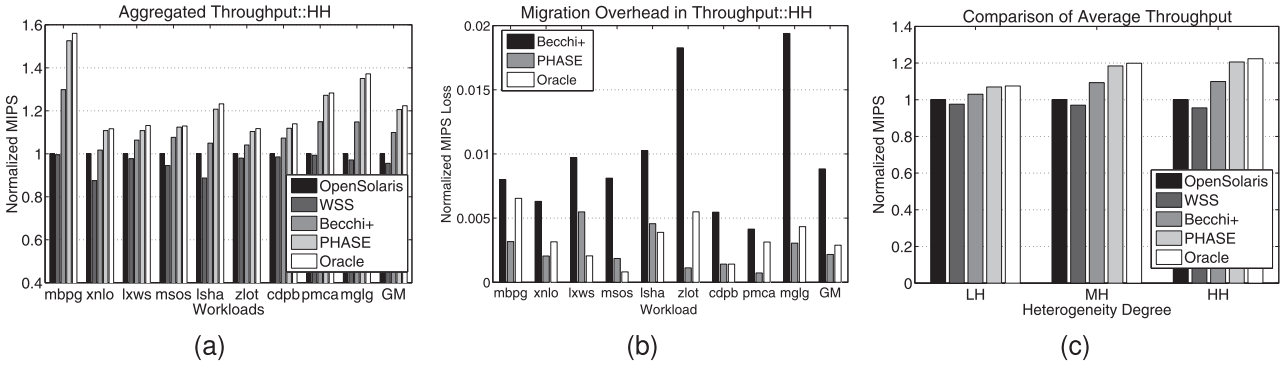

Fig. 7. Comparison of throughput. The data are normalized to the throughput of the OpenSolaris scheduler.

significantly higher (e.g., workload *xnlo*) than that of the *WSS* scheduler. This is because the *OpenSolaris* scheduler does not consider the underlying hardware heterogeneity, and the random nature of program-core assignment may end up with a reasonable good static assignment or the worst static assignment. This also means that a scheduler unaware of the core-level heterogeneity may lead to nondeterminist performance, which further underscores the importance of heterogeneity awareness in application schedulers. This figure also shows that *Becchi+* scheduler has significant improvement over the baseline *OpenSolaris* scheduler, yet its performance is still far from that of the *Oracle* scheduler due to its inability to quickly identify the optimum application-core assignment with explorative trial runs. In contrast, PHASE eliminates the trial runs and can achieve near optimum performance improvement. On average, it achieves 20.8 percent improvement over the baseline, 11.4 percent improvement over *Becchi+*, and is only 1.7 percent less than the oracle scheduling.

Fig. 7b illustrates the impact of migration overhead on the system throughput. It is obtained by comparing the realistic throughput with the throughput that is achieved when the data working sets are ideally moved along with the migrating applications. The migration overhead of *Becchi+* is consistently the largest for each workload due to the unnecessary movement of data sets and slows down the overall execution. Fig. 7c shows the average throughput (geometric mean) improvement as the heterogeneity degree changes. We observe that the potential of the throughput improvement drops as the heterogeneity degree decreases. This is because with reduced heterogeneity, the performance difference of scheduling an application to different cores is also reduced.

Improvement on efficiency. Fig. 8a shows the comparison of the efficiency in terms of $mips^3/W$ for different scheduling algorithms. We observe that PHASE achieves 3.2× efficiency improvement on workload *mbpg* compared with the baseline scheduler. This improvement is mainly because *OpenSolaris* scheduler blindly assigns the memory-bound *mcf* to the fastest core (C0) and the computing-bound *gcc* to the slowest core (C3), whereas PHASE schedules the programs in the opposite way, resulting high efficiency improvement. On average, PHASE improves the efficiency by 72.6 percent over the *OpenSolaris* scheduler and 37.2 percent over *Becchi+* scheduler. Note that for some workloads, such as *mbpg*, the *WSS* scheduler yields higher efficiency that the baseline scheduler, indicating that the baseline scheduler may require higher power consumption than *WSS* scheduler. Fig. 8b shows the efficiency loss caused by migration overhead. Again, *Becchi+* has the highest efficiency loss because the trial runs not only slow down the execution but also incur extra power consumption on the interconnection network between caches. Fig. 8c further shows the efficiency improvement as the heterogeneity level changes. Similarly, the potential of efficiency improvement decrease as the heterogeneity degree decreases.

Improvement on weighted speedup. Fig. 9 shows the performance and efficiency of different schedulers when using the weighted speedup as the optimization target. The results are similar with those of aggregated throughput, yet with smaller improvement. On average, PHASE improves

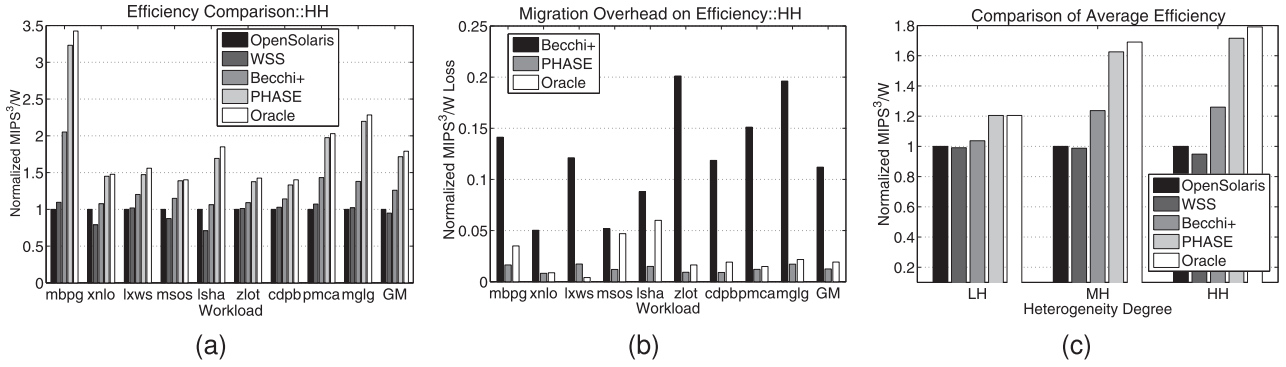

Fig. 8. Comparison of efficiency. The data are normalized to the efficiency of the OpenSolaris scheduler.

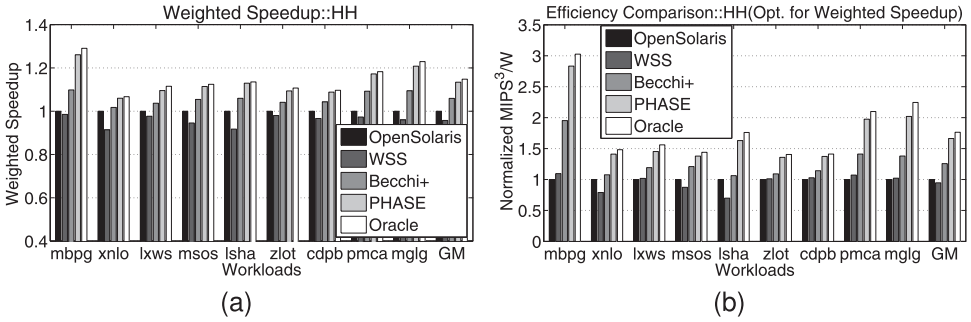

Fig. 9. Weighted speedup and efficiency.

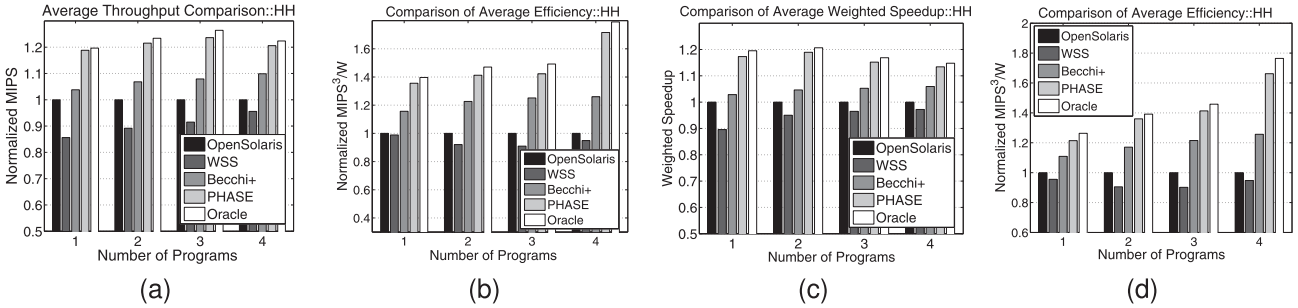

Fig. 10. Average performance and efficiency improvement as the program number changes.

the weighted speedup by 11.3 percent and the $mips^3/W$ efficiency by 59.7 percent over *OpenSolaris* scheduler, and compared with *Becchi+* scheduler, the improvements are 6.8 and 26.6 percent, respectively.

Impact of program number. We also evaluate the impact of program number on the performance of the schedulers. To do so, for each of the four-programmed workloads, we evaluate all possible combinations of 1, 2, and 3 programs. The geometric means of the throughput results are shown in Fig. 10a. Compared with the baseline scheduler, the performance of *Becchi+* decreases as the program number drops from 4 to 1. It is mainly because the scheduler unaware of the heterogeneity is more likely to reach a good static application assignment as the program number gets smaller. However, the performance of our predictive scheduling is still near optimum, and can reach up to 14.5 percent improvement over *Becchi+*. Fig. 10b further shows the efficiency improvement as the program number changes. Overall, the potential of efficiency improvement decreases as the number of program decreases. Figs. 10c and 10d show the results of the same experiment as Figs. 10a and 10b, but with weighted speedup as the optimization target. We observe the similar trend.

8.4 Scalability

Since PHASE depends on performance prediction for application scheduling, it does not require any trial runs and hence avoids the major scalability issue associated with the trial-and-error based dynamic scheduler. Meanwhile, to predict the application's performance, PHASE requires one hardware profiler per core, and hence the hardware cost of the framework scales only *linearly* with the number of cores. Another important aspect is the scalability of the searching algorithm that PHASE uses to identify the appropriate program-core allocation. To evaluate scalability of this algorithm, we measure its dynamic instruction count and its execution time under different number of cores. Fig. 11 shows the dynamic instruction count as well as the execution time of the searching algorithm. As we can see, even if the number of core reaches 32, it takes only 18.2 μs to execute the searching algorithm on a 4-GHz processor, which is less than 0.2 percent of the 10 ms scheduling interval and hence can be ignored. For CMPs with more than 32 cores, the scheduler could employ an hierarchical searching algorithm, which partitions the cores into groups of 32 cores or less and finds the appropriate program-core mapping hierarchically between groups and then within groups. In this way, the searching overhead remains low even for a large number of cores. Overall, PHASE demonstrates a good scalability with the number of core in the CMP system.

9 Related Work

Prior work on hardware-aware application scheduling can be classified into the following categories:

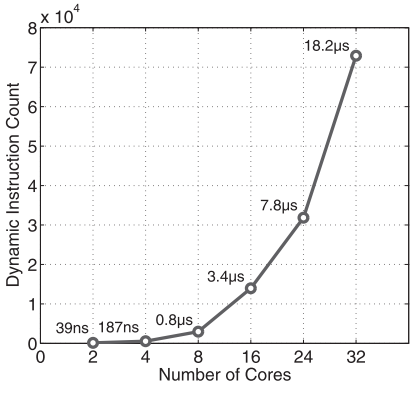

Fig. 11. Scalability of the searching algorithm.

Scheduling for single-ISA heterogeneous CMPs. Shepalov et al. [7] propose a heterogeneity-aware signature supported (HASS) scheduler, which relies on a signature generated by offline profiling to schedule the applications. Chen and John [8] proposed to use inherent program characteristics to find the proper program-core mapping offline. These offline schedulers cannot exploit dynamic phase changes, are sensitive to input data sets, and are impractical to implement without dramatic changes in the computer system. Kumar et.al [4] propose a dynamic scheduling scheme that tentatively runs application on different cores, and uses the sampled energy and performance data to find best application-core mapping. Becchi and Crowley [6] use the IPC ratio between the tentative runs on two different cores to migrate the application. These methods require trial runs, which not only incur overhead in power and performance, but also result in scalability issues. In contrast, our method does not require offline profiling nor any trial runs, and can achieve near optimum scheduling results.

Scheduling for on-chip variation. Teodorescu and Torellas [11] propose an application scheduler that is aware of intradie process variation on CMPs. This work only applies to cores with different frequencies and voltages but no other different microarchitectural features. In contrast, our PHASE scheduler addresses the scheduling challenge in a more heterogeneous CMP. Rangan et al. [29] propose Thread Motion, that involves migrating threads across processors with different voltage and frequency, as an alternative to DVFS. However, their architecture does not have any other source of heterogeneity and is effective only when multiple cores share an L1 cache. Recent work by Yan et al. [30] tries to address timing emergencies as a result of running multiple programs. They attempt to schedule workloads with large variations that may trigger timing emergencies on cores that can tolerate them. This is complementary to our work.

Scheduling for hard faults avoidance. Powell et al. [31] propose architectural core salvaging as a means for utilizing cores that cannot execute certain classes of instructions due to permanent faults. In this scheme, program migration is triggered only when a core runs into an instruction that cannot be executed. Therefore, this scheme is intended to ensure the execution correctness as opposed to enhance the performance, hence is complementary to our method.

10 Conclusions

As the transistor density and die sizes continue to grow, process variation and hard faults are expected to cause heterogeneity even in chip multiprocessors that were homogeneous by design. We show that there is the need to leverage such heterogeneity for application scheduling. However, the heterogeneity-aware schedulers proposed in the literature have inefficiencies and shortcomings, either causing significant overhead in power and performance or being impractical to implement.

This paper presents PHASE, a heterogeneity-aware scheduling framework that can dynamically and proactively schedule applications in single-ISA heterogeneous CMPs. This framework uses a set of hardware-efficient online profilers and an analytic performance model to simultaneously predict the application's performance on different cores. Based on the predicted performance, the scheduler identifies and enforces near optimum application assignment for each scheduling interval, eliminating the need of trial runs or offline profiling. We show that PHASE outperforms the OpenSolaris scheduler by an average of 20.8 percent in terms of overall throughput and an average of 72.6 percent in terms of efficiency. Compared with the state-of-the-art research scheduler, the proposed scheduler improves the throughput by an average of 11.4 percent and the efficiency by an average of 37.2 percent.

The proposed PHASE framework provides a platform which can be augmented to support additional structures as necessary. For example, it can be extended to support more manufacturing-caused heterogeneous resources, such as ROB sizes, or more coarse-grain heterogeneity from design, such as in-order and out-of-order execution styles. Overall, the proposed scheduling framework opens up a new possibility to leverage performance prediction to efficiently exploit what heterogeneous computing has to offer.

Acknowledgments

The authors would like to thank anonymous reviewers for the feedback. This work was supported partially through the US National Science Foundation (NSF) Award numbers 0702694 and 1117895. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of NSF.

References

[1] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter Variations and Impact on Circuits and Microarchitecture," *Proc. Design Automation Conf.*, pp. 338-342, 2003.

[2] P. Shivakumar, S.W. Keckler, C.R. Moore, and D. Burger, "Exploiting Microarchitectural Redundancy for Defect Tolerance," *Proc. Int'l Conf. Computer Design*, pp. 481-495, 2003.

[3] S. Ozdemir, D. Sinha, G. Memik, J. Adams, and H. Zhou, "Yield-Aware Cache Architectures," *Proc. Int'l Symp. Microarchitecture*, pp. 15-25, 2006.

[4] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen, "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," *Proc. Int'l Symp. Microarchitecture*, pp. 81-92, 2003.

[5] P.B. Daniel and M. Cesati, *Understanding the Linux Kernel*, chapter 7, third, ed. O'Reilly Media, 2005.

[6] M. Becchi and P. Crowley, "Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures," *Proc. Conf. Computing Frontiers*, pp. 29-40, 2006.

[7] D. Shelepov, J.C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z.F. Huang, S. Blagodurov, and V. Kumar, "HASS: A Scheduler for Heterogeneous Multicore Systems," *SIGOPS Operating Systems Rev.*, vol. 43, no. 2, pp. 66-75, 2009.

[8] J. Chen and L.K. John, "Efficient Program Scheduling for Heterogeneous Multi-Core Processors," *Proc. Design Automation Conf.*, pp. 927-930, 2009.

[9] "International Technology Roadmap for Semiconductors," http://public.itrs.net, 2006.

[10] R. Rao, D. Blaauw, D. Sylvester, and A. Devgan, "Modeling and Analysis of Parametric Yield under Power and Performance Constraints," *IEEE Design Test of Computers*, vol. 22, no. 4, pp. 376-385, July 2005.

[11] R. Teodorescu and J. Torrellas, "Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors," *Proc. Int'l Symp. Computer Architecture*, pp. 363-374, 2008.

[12] T.S. Karkhanis and J.E. Smith, "A First-Order Superscalar Processor Model," *Proc. Int'l Symp. Computer Architecture,* pp. 338-349, 2004.
[13] S. Eyerman, L. Eeckhout, T. Karkhanis, and J.E. Smith, "A Performance Counter Architecture for Computing Accurate CPI Components," *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems,* pp. 175-184, 2006.
[14] R.L. Mattson, D.R. Slutz, and I.L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems J.,* vol. 9, no. 2, pp. 78-117, 1970.
[15] B. Fields, S. Rubin, and R. Bodík, "Focusing Processor Policies via Critical-Path Prediction," *Proc. Int'l Symp. Computer Architecture,* pp. 74-85, 2001.
[16] M.D. Brown, J. Stark, and Y.N. Patt, "Select-Free Instruction Scheduling Logic," *Proc. Int'l Symp. Microarchitecture,* pp. 204-213, 2001.
[17] M.K. Qureshi and Y.N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," *Proc. Int'l Symp. Microarchitecture,* pp. 423-432, 2006.
[18] *Intel 64 and IA-32 Architectures Software Developer's Manual,* Volume 3B: System Programming Guide, chapter 7, O'Reilly Media, Sept. 2013.
[19] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer,* vol. 35, no. 2, pp. 50-58, 2002.
[20] M.M.K. Martin, D.J. Sorin, B.M. Beckmann, M.R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill, and D.A. Wood, "Multifacet's General Execution-Driven Multiprocessor Simulator (Gems) Toolset," *SIGARCH Computer Architecture News,* vol. 33, no. 4, pp. 92-99, 2005.
[21] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. Int'l Symp. Computer Architecture,* pp. 83-94, 2000.
[22] S. Thoziyoor, N. Muralimanohar, J.H. Ahn, and N.P. Jouppi, "Cacti 5.1," HP technical reports, pp. 1-37, 2008.
[23] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik, "Orion: A Power-Performance Simulator for Interconnection Networks," *Proc. Int'l Symp. Microarchitecture,* pp. 294-305, 2002.
[24] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, and J.D. Owens, "Memory Access Scheduling," *Proc. Int'l Symp. Computer Architecture,* pp. 128-138, 2000.
[25] "SPEC CPU2006 Benchmark Suit," http://www.spec.org, 2013.
[26] A. Phansalkar, A. Joshi, and L.K. John, "Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite," *Proc. Int'l Symp. Computer Architecture,* pp. 338-349, 2007.
[27] A. Snavely and D.M. Tullsen, "Symbiotic Job Scheduling for a Simultaneous Multithreaded Processor," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 234-244, 2000.
[28] D.M. Brooks, P. Bose, S.E. Schuster, H. Jacobson, P.N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P.W. Cook, "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors," *IEEE Micro,* vol. 20, no. 6, pp. 26-44, Nov./Dec. 2000.
[29] K.K. Rangan, G.-Y. Wei, and D. Brooks, "Thread Motion: Fine-Grained Power Management for Multi-Core Systems," *Proc. Int'l Symp. Computer Architecture,* pp. 302-313, 2009.
[30] G. Yan, X. Liang, Y. Han, and X. Li, "Leveraging the Core-Level Complementary Effects of PVT Variations to Reduce Timing Emergencies in Multi-Core Processors," *Proc. Int'l Symp. Computer Architecture,* pp. 485-496, 2010.
[31] M.D. Powell, A. Biswas, S. Gupta, and S.S. Mukherjee, "Architectural Core Salvaging in a Multi-Core Processor for Hard-Error Tolerance," *Proc. Int'l Symp. Computer Architecture,* pp. 93-104, 2009.

Jian Chen received the BE and ME degrees in electrical engineering from Shanghai Jiao Tong University, in 2002 and 2005, respectively, and the PhD degree in computer engineering from The University of Texas at Austin in 2011. He is currently a performance architect in Intel Corporation. His research interests include computer architecture, workload characterization and performance modeling. He is a member of the IEEE, the IEEE Computer Society, and ACM.

Arun Arvind Nair received the BE degree in electronics engineering from the University of Mumbai, India, in 2002, the MS degree in computer engineering from the University of California at Irvine in 2006, and the PhD degree in computer engineering from The University of Texas at Austin in 2012. He is currently a performance architect in AMD. His research interests include architectural techniques for reliability, workload characterization, and analytical modeling. He is a member of the IEEE and the IEEE Computer Society.

Lizy K. John received the PhD degree in computer engineering from The Pennsylvania State University in 1993. She currently holds the B.N. Gafford Professorship in the Electrical Communication Engineering Department at The University of Texas at Austin. Her research interests include microprocessor architecture, performance and power modeling, workload characterization, and low power architecture. She is a fellow of the IEEE and the IEEE Computer Society.