FPS: A Fair-Progress Process Scheduling Policy on Shared-Memory Multiprocessors

Chenggang Wu, Jin Li, Di Xu, Pen-Chung Yew, *Fellow, IEEE*, Jianjun Li, and Zhenjiang Wang

Abstract—Competition for shared memory resources on multiprocessors is the dominant cause for slowing down applications and making their performance varies unpredictably. It exacerbates the need for Quality of Service (QoS) on such systems. In this paper, we propose a *fair-progress process scheduling* (FPS) policy to improve system fairness. The strategy is to force the equally-weighted applications to bear the same amount of slowdown when they run concurrently. When we find an application suffered more slowdown and accumulated less effective work than others, we allocate more CPU time to give it a better parity. This policy can also be applied to threads with different weights. Evaluation results show that FPS can significantly improve system fairness at the expense of a slight loss in throughput. We can also keep the performance information of an application to guide process scheduling when it runs again later on. When FPS uses such performance information from previous runs, fairness can be maintained without the overhead of the training periods required in FPS. Throughput can thus be enhanced.

Index Terms—Cross-run optimization, memory bandwidth, process scheduling, performance fairness.

1 Introduction

Shared-memory multi-core processors are the most prevalent platforms used today. When applications run concurrently on such system, competition for shared memory resources such as on-chip caches and DRAM subsystems could degrade their performance unpredictably (compared to when they run alone on the same system). Fig. 1 shows the effect of resource contention on four equally-weighted and concurrently running applications, *perl*, *bwaves*, *milc* and *libquantum*, all from SPEC2006. They run on a 4-core CMP with private cache (details are in Section 3.1). Compared to the isolated run, the execution time of *perl* increases by 1.10×, while the execution time of *libquantum* increases by 1.62× because *libquantum* is memory-intensive. It suffers more slowdown due to off-chip main memory contention. If we replace the last two co-runners with *leslie3d* and *soplex*, the relative slowdown of *bwaves* changes from 1.19× to 1.47×. It shows that performance of an application highly depends on its co-runners and can change unpredictably due to resource contention. It violates the assumption of the weight-based CPU time allocation policy in the OS, and exacerbates the need for Quality of Service (QoS) on such systems.

- *C. Wu, J. Li, and Z. Wang are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, P.R. China. E-mail: {wucg, lijianjun, wangzhenjiang}@ict.ac.cn.*
- *J. Li is with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, and with the University of Chinese Academy of Sciences, Beijing 100190, P.R. China. E-mail: lijin@ict.ac.cn.*
- *D. Xu is now with IBM Research-China, Beijing, China. E-mail: xudi@cn.ibm.com.*
- *P.-C. Yew is with the Department of Computer Science and Engineering, University of Minnesota at Twin-Cities, Minnesota, MN 55455. E-mail: yew@cs.umn.edu.*

Manuscript received 15 May 2013; revised 23 Dec. 2013; accepted 30 Dec. 2013. Date of publication 12 Mar. 2014; date of current version 9 Jan. 2015. Recommended for acceptance by I. Stojmenovic.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2306411

In order to provide better fairness to concurrently running applications, some prior work tried to guarantee the applications their fair share of system resources, such as cache space [8] and/or memory bandwidth [10]. Some tried to maintain similar performance on demanded resources, such as cache miss rates [3], [21] and memory-related stall time [6]. However, there still exist gaps between the share of demanded resources or the resource performance and the real application performance (e.g., IPC). In this paper, we assume that, for equally-weighted applications, a system is *fair* if all applications' experienced slowdowns are the same. This assumption is based on *application performance* rather than on *resource-related metrics*. It has also been used in prior work [2], [9], [11].

In this paper, we propose a *fair-progress scheduling* (FPS) policy, a process scheduling policy that ensures fairness among applications running concurrently. Its basic mechanism is as follows. For each running application, we use the data gathered from the performance monitoring unit (PMU) and the results from an analytical model to derive the amount of its *forward progress* after the extent of a time quantum. If we find the application has suffered more slowdown (thus accumulated less progress) than others within the time quantum, we allocate more time quanta to the application and allow it to make the same *forward progress* as others.

To calculate the *forward progress* of an application, the main challenge is to estimate its performance if it runs alone on the system while it is actually running simultaneously with others [2], [6]. In this paper, we propose a software-based approach. First, we aggregate all executed time quanta into phases [15], [16], [17], [18]. The performance in the time quanta of the same phase should be very similar. In each phase, if the memory bandwidth contention in one quantum meets one of the three *low-contention* criterions (Section 3.1), its performance is close to its run-alone performance. We then incorporate the information to other time quanta in the same phase and estimate their *forward progress*.

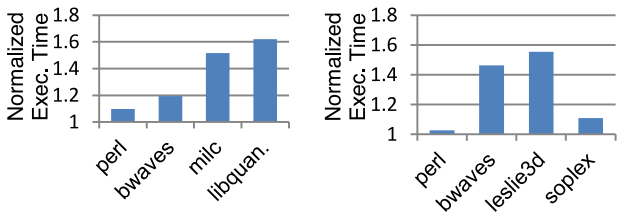


Fig. 1. Performance variations of concurrent applications.

For a phase without such *low-contention* quanta, we insert a *training quantum* in the phase with the desired *low-contention* by reducing the number of co-runners during the training quantum (Section 3.3). Training improves the accuracy of run-alone performance estimation and the system fairness at the expense of some resource idleness and throughput degradation. To mitigate this problem, we propose techniques to effectively reduce such training overhead. In addition, by setting an upper bound on the training overhead, FPS gives the OS an effective and robust mechanism to balance between system fairness and throughput. It also allows us to achieve different fairness objectives.

In modern computer systems, such as data centers, an application often runs multiple times, each maybe with different co-runners. In such scenarios, we found it is feasible to utilize the information of previous runs to reduce the training overhead and throughput degradation in later runs. The reason is that identical phase identified by our phase identification scheme has similar performance in each run regardless of which their co-runners are. Hence, such performance information can be used across multiple runs.

We propose phase-based *cross-run* optimization that uses a *phase repository* to keep phase information across program executions. This mechanism can be fully automatic and transparent to the users. Information acquired during *low-contention* time quanta can be accumulated over different runs. The need of inserting training quanta can be reduced, and so is the degradation of system throughput.

Our experiments are done on commercial servers with Intel multicore processors. Our results show that compared to a native OS scheduler, FPS can improve system fairness by 53.5 and 65.2 percent on a 4-core system with private cache memories and a 4-core system with a shared cache memory with throughput degradation of 1.1 and 1.6 percent, respectively. For memory-intensive workloads, FPS improves system fairness by an average of 45.2 and 21.1 percent on 4-core and 8-core systems, respectively when capping the training overhead to 2 percent. Even without using any training, FPS can still get a 15 percent fairness improvement over the default OS scheduler with negligible throughput degradation. When cross-run optimization applied to FPS, the overhead and system throughput degradation brought by training have been eliminated dramatically at the cost of some extra temporary disk space per application.

The main contributions of this work are as follows.

First, we propose a scheduling policy to provide performance fairness on commodity multicore systems. The policy can significantly improve system fairness at the expense of slightly decreased throughput.

Second, we propose a practical, phase-based run-time scheme to obtain the run-alone performance of an application while it actually runs concurrently with others. It is software-based and does not need any special hardware support.

Third, an effective and robust tuning scheme is used to let the user freely make tradeoffs between system fairness and system throughput.

Compared to our previous work [23], we introduce an automatic and transparent cross-run optimization scheme that accumulates phase information across multiple runs to alleviate the need of training quanta.

The rest of this paper is organized as follows. Section 2 shows an overview of our fairness policy. The run-alone-performance estimation scheme is introduced in Section 3. Section 4 introduces the cross-run optimization scheme. Sections 5 and 6 present the evaluation methodology and results, respectively. Related works and discussions are present in Section 7. Finally, we conclude our work in Section 8.

2 Policy Overview

2.1 Specifying the Fairness Target

Similar to previous work [2], [6], [9], [11], we assume a system is *fair* if equally-weighted applications experience same amount of slowdown when they run concurrently on the system. As shown in (1) and (2), on a system with N applications, the slowdown of application i is T^i_{shared}/T^i_{alone}, where T^i_{shared} and T^i_{alone} are the execution time when the application runs concurrently with others and runs alone, respectively. In the context of process scheduling, an application's execution time T includes both the time when it executes on a CPU and the time when it is swapped out. System *unfairness* is defined as the ratio between the maximal and minimal slowdown among the N applications. An unfairness of 1 means the system is perfectly fair

$$slowdown_i = T^i_{shared}/T^i_{alone} \tag{1}$$

$$unfairness = \frac{MAX\{slowdown_0, \ldots, slowdown_{N-1}\}}{MIN\{slowdown_0, \ldots, slowdown_{N-1}\}}. \tag{2}$$

2.2 Basic Ideas

FPS tries to guarantee that equally-weighted applications experience the same *slowdown* when they run concurrently. In another word, the applications should accomplish the same amount of *effective work* (measured in T_{alone}) in the same time period. We use forward *progress* to quantitatively measure the *effective work* that an application has completed. Assume when an application runs alone for C_{unit} cycles, it makes a progress of 1. When it runs concurrently with others, its *progress* can be calculated as follows:

$$progress = \sum_{q=1}^{Q} \frac{C^q_{alone}}{C_{unit}} = \sum_{q=1}^{Q} \left(\frac{I^q}{IPC^q_{alone}} \times \frac{1}{C_{unit}} \right). \tag{3}$$

For a time quantum q, C^q_{alone} is the number of CPU cycles needed when an application runs alone. An application's *progress* is the accumulation of C^q_{alone} in each executed time quantum (from 1 to Q) normalized toC_{unit}. For example, an application runs simultaneously with others for C_{unit}*2 cycles but suffers 4× slowdown, it has only made a *progress*

of 0.5, while another application runs for C_{unit} cycles but does not have any slowdown, it has made a *progress* of 1.

In (3), I^q is the number of executed instructions during time quantum q. It can be obtained directly from PMU. The main challenge is to estimate IPC^q_{alone}. In Section 3, we describe our proposed method in more detail.

In order to enforce equally-weighted applications to have same amount of slowdown, FPS tries to let them achieve the same *progress* in a given time period. Algorithm 1 shows the basic scheme of FPS. Each time we need to schedule applications, we update the *progress* of each application according to the runtime information gathered from PMU and the estimated IPC^q_{alone}. We then schedule the application with the smallest progress first on each available CPU core.

Algorithm 1: Overview of Policy

```
01  Initialize task-queue: run-queue, wait-queue
02  while task remains do
03    run all apps in run-queue
04    wait for a quantum
05    // apps run, and then the scheduler resumes here:
06    for each app in run-queue do
07      pause execution
08      move app to wait-queue
09      process with PMU data
10      estimate IPC_alone of this quantum
11      progress+= I/IPC_alone × 1/C_unit
12    end for
13    // schedule for fairness:
14    for each available CPU core do
15      find app with smallest progress in wait-queue
16      move app to run-queue
17    end for
18  end while
```

3 Estimating Run-Alone Performance

To estimate the run-alone performance in each time quantum, we make use of the phase behavior in applications. First, we group executed quanta into *phases*, and use the attribute that the performance of the quanta in the same phase should be similar [15], [16], [17], [18]. If we know IPC^q_{alone} of at least one quantum q in a phase, we can apply the information to other quanta in the same phase. How to estimate IPC_{alone} of a given phase? An intuitive solution is to select some quanta in that phase and let them run alone. But, this method would result in large CPU idleness if there are many applications and each has many different phases. Fortunately, we found that in some situations, even when an application run concurrently with others, its performance is still the same as (or very similar to) that of when it runs alone. In which case, we can get the estimated IPC^q_{alone} without running the application alone.

3.1 Identifying *Low-Contention* Applications in a Quantum

Competition for shared memory resource is the primary cause for performance variations [2], [3], [6], [7], [8], [21], [22]. Even if contention happens in an unpredictable way, we observed that at least in three cases, the execution of an application suffers little or minor interference. Hence, we could assume that $IPC^q_{alone} \approx IPC^q_{shared}$ in those cases.

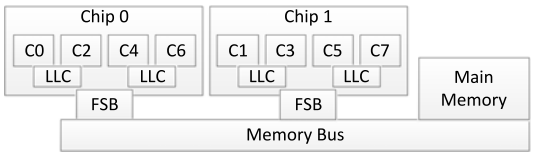


Fig. 2. Architecture overview of the evaluation system.

In this section, we use measured results on real systems to better explain the phenomenon. Fig. 2 shows the evaluation system. The system is equipped with two Intel Xeon E5410 quad-core processors. Each core has a private L1 data cache, and each two cores share one L2 data cache (LLC) on the chip. The benchmarks are from SPEC2006 suite, compiled by Intel Compiler with flag-O3, and use the *reference input set*. We generate 10 random benchmark mixes. To evaluate the contention on the main memory, we run benchmarks on cores 0, 1, 4 and 5 to isolate the impact of the shared cache. We also evaluate the situation in which both cache and main memory are shared by executing them on cores 0, 1, 2 and 3. We use *Bus Transaction Rate (BTR)* to characterize the memory-bandwidth requirement of the execution. BTR is defined as the number of full-cache-line bus transactions per microsecond. The realistic peak BTR of the memory bus is 120 trans./usec, and that of the FSB is 80 trans./usec. The OS is Linux, kernel version v2.6.29.

In this paper, we call the execution part of an application during a scheduling time quantum an *application segment*, or *segment* for short. In each segment, its IPC^q_{shared} can be obtained via PMU. We also get its IPC^q_{alone} by querying an offline performance profiling file, which is generated by executing the application alone on the same system. At last we can calculate its speedup as $IPC^q_{shared}/IPC^q_{alone}$. Note the method of obtaining IPC^q_{alone} by profiling is only for our evaluation and analysis purpose. It is not a part of our scheduling policy.

Memory bandwidth contention. We associate the speedup of each segment s with two memory-bandwidth related metrics: $SelfBTR_s$, i.e., the BTR of s when running with others, and $SysWideBTR_s$, i.e., the total BTR of s and all of its co-runners within the quantum. To show the correlation among the metrics, Figs. 3a and 3b plot the evaluation results of the same randomly selected and representative 1k segments in private-cache mode. Fig. 3c shows a subset of them. The three *low-contention* cases are as follows:

1. The bandwidth requirement of a segment is extremely low (Criterion 1). In this case, its performance degradation due to bandwidth contention is very small no matter what applications it runs with. Evaluation results shown in Fig. 3a confirm that memory-intensive segments generally suffer more slowdown than less intensive ones [1]. In this paper, we set a threshold $BTR_{SelfLow}$ to 4 percent of the peak BTR. If BTR of a segment is smaller than the threshold, we assume its IPC^q_{shared} is similar to IPC^q_{alone}.

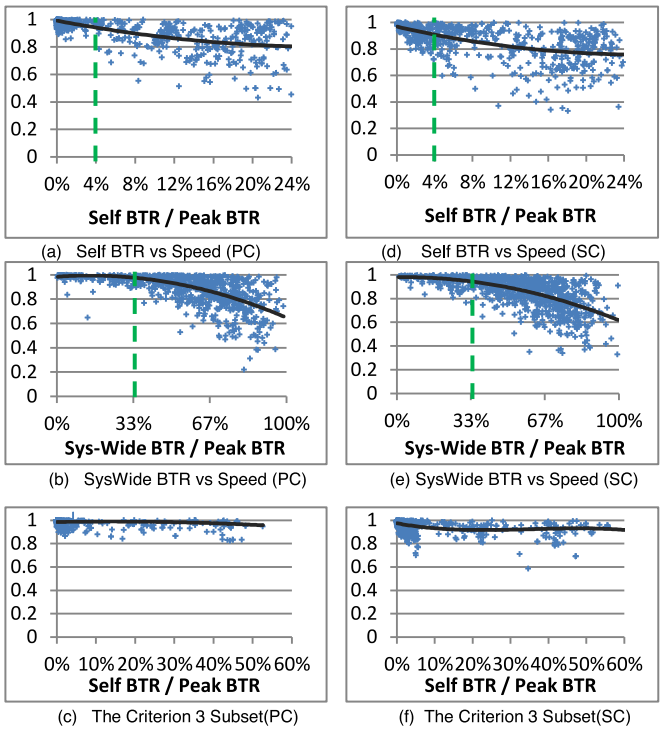


Fig. 3. Speedup over alone run on private cache mode (PC) and shared cache mode (SC). X-axes are the percentage of measured BTR compared to peak BTR. Y-axes are speedups, 1 means no slowdown. The solid lines show the curve-fitting of the points. The dashed lines illustrate the thresholds.

2. The system-wide bandwidth utilization is low (Criterion 2). In [4], authors found that even when the average bandwidth requirement of concurrent segments is lower than the realistic peak bus bandwidth, contentions still happen because of the fluctuation in memory intensity within the segments. Only when the system bandwidth utilization is much lower than peak, the remaining available bandwidth could tolerate the fine-grained contention and that results in a relatively smaller slowdown. In this paper, we set a system-wide BTR threshold $BTR_{SyswideLow}$ to be one third of the peak BTR. If the system-wide BTR is smaller than $BTR_{SyswideLow}$, we assume IPC^q_{shared} is similar to IPC^q_{alone} for all the concurrent segments.
3. The system wide BTR is larger than $BTR_{SyswideLow}$ but only one of the concurrent segments is memory-intensive, i.e., whose BTR is larger than $BTR_{SelfLow}$ (Criterion 3). Severe contention only happens when there are at least two memory intensive segments that are fighting against each other. Fig. 3c shows a subset of evaluated segments that belong to this case. Compare it with Fig. 3a, we found using this criterion could successfully pick up those quanta that have inherently high memory requirement but suffer relatively small slowdown.

Cache and Bandwidth Contention. Similarly, Figs. 3d, 3e, 3f show the execution results in shared-cache mode. Compared to the results in private-cache mode, we have the following observations. First, the performance degradation could become more severe because of the added shared cache contention. Second, the tendency of performance variations is quite similar to that of the private-cache mode. Previous work also shows that contention for shared cache is not the dominant cause for performance degradation, but the contention in many components of the main-memory is [1]. Although the segments selected according to the three abovementioned criteria have bigger performance degradation compared to that in the private-cache mode, their performance degradation is still relatively small. As will be shown later, the information gathered from such co-scheduled applications will help to estimate the IPC^q_{alone} in each executed quantum.

3.2 Phase Identification and Performance Information Management

Program phase behavior has been studied extensively. In this paper, we use a runtime *basic block vector (BBV)-based* phase identification scheme [15], [16], [17], [18] to classify similarly behaved quanta into a *phase*. BBV analysis has been shown to be an effective method of identifying phases in programs [15], [16].

During the execution of each application segment, we use PMU to sample its instruction pointers (IP) and construct a BBV. Each element of the BBV maps to a static basic block, and its value is increased by 1 when an IP sample falls into it, so the BBV reflects the distribution of the sampled IPs in the application's code space. We quantify the similarity of two normalized BBVs by calculating their Manhattan Distance, as shown in (4) with x_i and y_i each an element of vector X and Y, respectively. N is the number of static basic blocks in the application binary. If the Manhattan Distance of two BBVs is smaller than a threshold, the corresponding segments are in the same phase

$$ManhattanDistance(X,Y) = \sum_{i=1}^{N} |x_i - y_i|. \quad (4)$$

Evaluation results of PMU sampling overhead and phase identification accuracy are in supplemental materials, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2014.2306411.

For each application, we maintain a *phase table* to record the performance information of its phases. The goal is to estimate the *run-alone performance* of a given phase, denoted as IPC_{phase}. After the execution of a quantum q, first we do phase identification and calculate some basic performance metrics such as BTR^q_{shared}, IPC^q_{shared} and system-wide BTR according to the PMU data. We then check whether the application's execution meets one of the three *low-contention* criteria. If it does, its performance data is considered *valid*, otherwise *invalid*.

In any case, we update the performance information in the phase table following the method shown in Table 1. Before a phase gets its first valid execution, IPC_{phase} is the average of all previous invalid executions. After the phase gets its first valid execution, the intended value of IPC_{phase} is the average of all previous valid executions. The method provides the phase with a running estimate, and the three *low-contention* criteria work as a filter to let only suitable data participate in the estimation of IPC_{phase}.

TABLE 1
Per-Phase Performance Info Update Method

Exec. history	IPC^q_{shared} of current quantum	Updated IPC_{phase}
None	Valid/Invalid	$IPC_{phase} = IPC^q_{shared}$
#valid=0	Invalid	$IPC_{phase} = (IPC_{phase} * n + IPC^q_{shared})/(n+1)$
	Valid	$IPC_{phase} = IPC^q_{shared}$
#valid >=1	Invalid	*Unchanged*
	Valid	$IPC_{phase} = (IPC_{phase} * n + IPC^q_{shared})/(n+1)$

Finally, the estimated IPC^q_{alone} for this quantum is the larger one between IPC_{phase} and IPC^q_{shared}, because we assume the IPC of a simultaneously running segment cannot be larger than that of when it runs alone.

3.3 Training Quantum

During execution, a phase may never have a quantum that fits any of the three *low-contention* cases. It can degrade the accuracy of the estimation. In such a case, we can inject a *training quantum* to create a *low-contention* situation in the phase.

For each application, we use a Markov predictor [17] to predict its phase changes based on its full history. To tolerate incorrect prediction, the predictor reports several phases that are most likely to appear in next quantum according to their transfer probabilities. If one of the predicted phases has appeared at least twice in a history and we have not obtained its valid IPC_{alone} yet, that application will become a training-target candidate. A training target selection policy is used to determine a target if there are multiple potential targets, then FPS transfers the scheduling policy to *training* in next quantum. More details about training are present in supplemental materials, which is available online.

4 FPS with Cross-Run Optimization

4.1 Cross-Run Approach

So far we assume training is done *each time* an application runs on the system. However, on modern computer systems such as data centers, applications often run repeatedly with different co-runners and different input sets. In such scenarios, even though the overhead and the degradation in system throughput brought by training is not high in each execution, it becomes noticeable when the application runs repeatedly and such overheads are sustained. We can use *fairness-throughput tuner* to make a tradeoff between system fairness and throughput. It mitigates this problem but not eliminates it. In addition, the phase table is discarded at the end of one execution. It needs to be reconstructed from scratch in the next run, wasting the valuable information obtained in the previous runs. In this section, we propose to capitalize on such information and to improve FPS across different runs. It uses a phase repository to allow such phase information to persist across runs for an application.

As described in Section 3.2, we use a runtime *basic block vector* (*BBV*)-based phase identification scheme to group similarly behaved quanta into phases. The identical phase recognized by BBV in different runs has similar behavior due to it always represents same sections of code and same amount of time spent in this code. The intuition behind this is that the behavior of program at a given time is directly related to the code it is executing and its frequency during that interval [15]. It is independent of any individual architecture metric [17]. In FPS, valid IPC_{phase} information of a phase is discarded when the application exits. When it appears in another run, training quanta could occur again.

So we can aggregate the valid phase information in the phase table into a phase repository called *phasedb* at the end of each execution. The format of an entry in *phasedb* is same to that in phase table. Phase table records runtime phase information in memory including valid and invalid while *phasedb* preserves only valid phase information accumulated from previous runs in disk. Each application has its own *phasedb* with a prefix of its name. When an application runs next time, it first reads phase information from its *phasedb* into its runtime corresponding phase table and then updates it during its execution using the same method described in Section 3.2. Reading and writing from the repository doesn't take up the execution time of an application and is optional. So, it doesn't bring extra time cost to the FPS. If we don't want to use cross-run optimization, it can simply be ignored.

In this way, training quanta are injected only at the first few runs. As valid information accumulated, training quanta reduced greatly in the later runs. So the throughput degradation only exists at the beginning and can be ignored during the multiple runs.

The only cost brought by the cross-run optimization is some extra temporary disk space. The total disk space required is modest because only valid phase information is accumulated and stored. It can also be cleared or taken away at any time.

4.2 Sensitivity to Input Sets

A program input set refers to all of the data that are accessed but not generated during a program execution, including command-line options, content of input files, and so on [27].

Program inputs may change the code sections being executed and/or its executing frequency. That could make FPS get different samples of BBV. As a result, new phases and phase sequences could be brought in when input changed. Some of the valid phase information can still be used if that phase appears in the other run, while some of it may not. That depends on how much change that input has brought. However, what we care is the new phases that brought by the input set change rather than the input set itself. Some new phases can get valid IPC_{alone} from spontaneous *low-contention* execution. If that does not happen, it would bring training quanta. The number of training quanta will not be greater than the situation when cross run optimization doesn't be applied to FPS. So, cross-run would not bring additional overhead. In addition, input doesn't affect the system fairness benefit brought by FPS at all. When this input appears again in the future run, the number of training quanta would reduce a lot because valid phase information has been accumulated in the phase repository.

TABLE 2
Parameters Setup of Scheduler

Scheduling quantum length	100ms
PMU sampling period	500k instructions
BBV similarity threshold	0.7
Training overhead limitation	Unlimited
Sequential training trigger	#failed training>5 && failure ratio>60%

5 Methodology

Scheduler implementation. In order to evaluate the effectiveness of our FPS scheduling policy, we implemented a user-level process scheduler in Linux. The scheduler itself executes as a daemon. When an application is scheduled to run, the scheduler forks out a process and creates a PMU sampling context for it. The scheduler sets a timer to count the scheduling time quanta. When the timer expires, the scheduler is notified and enters its scheduling routine. The algorithm is described in Section 2.2. Applications are paused by PTRACE_ATTACH, so the Linux kernel will not schedule them. We release the selected applications using PTRACE_DETACH and let them run, reset the timer, and let the scheduler sleep to wait for the next notification.

System setup. We use the same system setup described in Section 3.1. Table 1 in supplemental materials, which is available online, shows the system setups for our evaluations. When evaluated on a 4-core system, we use different CPU cores to create different scenarios of resource contention. We also evaluated on an 8-core system. Unless stated otherwise, Table 2 shows the parameters used in our evaluation.

Workloads. Workloads are constructed using SPEC CPU 2006 suite. We run each benchmark alone for 10 seconds, and record the number of instructions executed. The same part of execution is used when a benchmark is selected to run. Job normalization eliminates the variation of the execution length. It results in a fair measure of the potential unfairness because almost all parts of benchmarks are running concurrently. It also gives fair measurements of system throughput because if the applications are different in length, load balancing could influence the results significantly [4].

To evaluate the generality of FPS, we use 10 randomly generated multiprogramming workloads that are presented in details in the supplemental materials, which is available online.

Metrics. We compare the results of FPS to the native Linux scheduler (kernel v2.6). In order to show fairness improvement, we measure system *unfairness* as defined in Section 2.1. We use two metrics to evaluate the impact of FPS to system throughput: *work-load turnaround time*, i.e., the time from when all applications start at the same time to the last application finishes; and *extended weighted speedup (EWSpeedup)*. The weighted speedup defined in [11] has been used to measure the throughput of multiprocessors systems on which the number of concurrent threads does not exceed the number of CPU cores. It is calculated as the *sum* of the speedups of all concurrent applications. However, a job scheduling policy may pathologically improve this metric by forcing all jobs run serially so that each job suffers no slowdown. So, we define *EWSpeedup* as shown in

$$EWSpeedup = \sum_{i=0}^{N-1} \frac{C^i_{alone}}{C^i_{shared} + C^i_{wasted}}. \quad (5)$$

The key to the definition is that, when calculating the speedup of an application i, if any CPU core becomes idle because the number of remaining applications becomes smaller than the number of available cores, the wasted cycles on the idle CPU core are counted as its execution time. It is because in our policy, training causes CPU to become idle, and we attribute such wasted cycles to the training target in the corresponding training quantum. A *harmonic mean* is usually used for *speedups* [14] to give a combined measure of both fairness and throughput. We don't use this metric because it is determined by applications' individual IPCs. Our process scheduling is not to give the applications fair IPC during execution, but to adjust CPU time distribution to achieve fair execution time.

To give a thorough demonstration of the effectiveness of cross-run optimization, we show that identical phases recognized by BBV have similar performance even when the input changes. We run a program alone to record its phase information such as IPC_{alone} and keep it in a phase repository. In the following runs, if BBV recognizes one phase has appeared in a previous run, we use IBM SPSS statistics tools to analyze whether the stored phase information is applicable to this run. The evaluation results are presented in the supplemental materials (Section 5), which is available online.

Then we evaluate system performance using the following usage scenarios.

Scenario 1 comparison. We use this scenario to compare the previous results of FPS. Each workload runs 20 times from scratch with fixed input under reference workload and gets an average result. Then an average result of all the 10 workloads represents the results on one platform. In this way, cross-run is actually applied within one workload. Evaluation results are shown in Sections 6.1, 6.2 and 6.3.

Scenario 2 random co-runners. This scenario is used to show valid phase that preserved in one run can be used in another run even though the co-runners have changed. All the 10 workloads run in a randomly selected order. The total 20 times executions represent the random execution sequences of all the workloads. Evaluation result is shown in Section 6.5.

Scenario 3 variable inputs. This scenario is used to show the impact of input changes on FPS and cross-run optimization. We extend WL#9 to make it contain all the benchmarks that contain multiple input files. Then we run WL #9 20 times, benchmarks with multiple input files change an input in a round-robin manner each time. The reason we extend WL #9 is it has already contained most of these benchmarks and has a larger IABW when extended. We can get similar results if we chose to extend other workloads. Evaluation result is shown in Section 6.6.

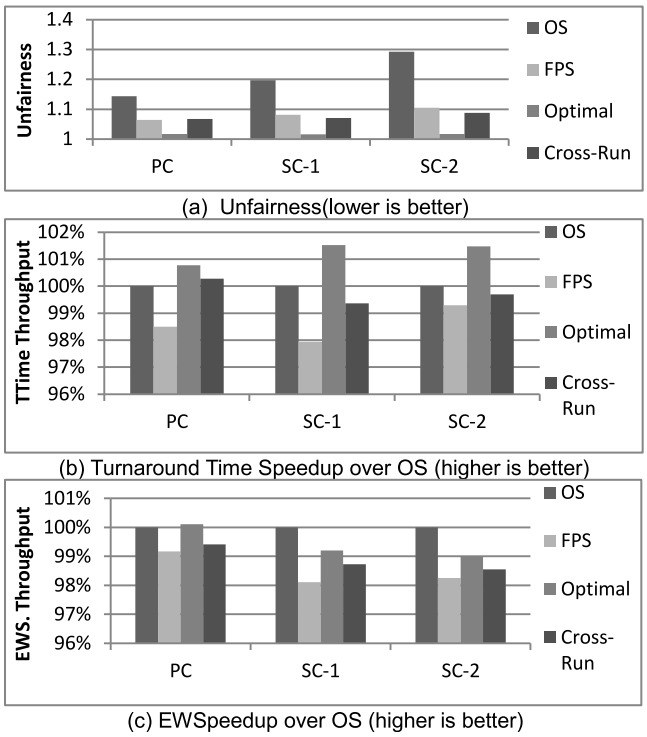


Fig. 4. Average performance on 4-core systems.

For each scenario, the system fairness benefit doesn't lose, however, the training overhead and the degradation of system throughput has greatly reduced.

6 Evaluation Results

6.1 4-Core System Results

Fig. 4 shows the average performance of 10 random workloads on the 4-core systems. We evaluate four different scheduling policies: the native OS, FPS, and Optimal. Cross-run represents cross-run optimization applied to FPS. The optimal policy uses our fairness-oriented scheduling algorithm but IPC_{alone} of each quantum comes from offline profiling runs instead of runtime estimation, as described in Section 3.1.

In the private-cache mode, the unfairness under OS scheduler ranges from 1.08 to 1.23, the average is 1.14. FPS improves fairness on all workloads. The average of unfairness is decreased to 1.06, and about 53.5 percent of the unfairness is eliminated. On the shared-cache-diff-chip mode, the system peak bandwidth is unchanged but cache contention is added. Cache contention and the resulted higher pressure to bandwidth let the system unfairness under OS increased to 1.22. About 65.0 percent of the unfairness is eliminated by FPS, and it decreased to 1.08. On the shared-cache-same-chip mode, bandwidth contention is further increased. The unfairness under OS is decreased to 1.29. The unfairness under FPS is 1.10. FPS eliminates about 65.8 percent of the unfairness. Results show that FPS is effective in eliminating unfairness on both private-and shared-cache systems. The unfairness results of cross-run optimization are close to the FPS in both private and shared-cache systems.

For all the evaluated modes, the most severe system throughput degradation is about 2 percent compared to

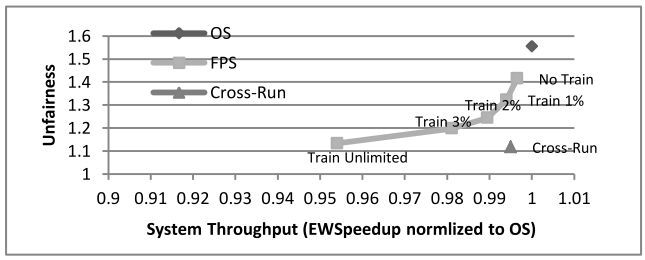


Fig. 5. Effect of different training limitation.

OS, no matter whether it was measured in workload turnaround time or EWSpeedup. The average decrease is 1.1, 2.06 and 1.12 percent respectively for the three execution modes. The main sources of the degradation is PMU sampling and CPU idleness caused by training. Cross-run optimization can eliminate the degradation brought by training and the average decrease is no more than 1 percent.

In every situation, the optimal policy achieves almost perfect fairness on all workloads and the best throughput. It shows the effectiveness of the fairness-oriented scheduling algorithm.

6.2 8-Core System Results

Fig. 5 shows the average performance of the 10 workloads on 8-core system. System unfairness under the native OS increases to 1.55 compared to the 4-core modes. FPS could eliminate about 15 percent of the unfairness even without any training, because it uses the data from spontaneous *low-contention* execution. The overhead is quite negligible. When we gradually relax the training overhead limitation, the fairness improves significantly and system throughput degrades slightly. When the training overhead is unlimited, FPS eliminates 70 percent of the system unfairness at the expense of 4.5 percent system throughput degradation. Compared to the results on 4-core systems, throughput degradation is slightly larger because the number of available CPU cores is doubled but the training parallelism is nearly unchanged. Hence, more CPU idleness is incurred due to training. Evaluation shows that our fairness-throughput tuner is a tradeoff between fairness and system throughput.

Tuner (Section 2 in the supplemental materials, which is available online) brings moderate performance to the system. It is just an approach of mitigation rather than elimination. However, cross-run optimization can handle this problem thoroughly. When it applied to FPS, system unfairness is decreased to 1.12, eliminating about 75 percent of the unfairness. At the same time, the system throughput degradation is only about 0.5 percent.

We use *train ratio* to represent the percentage of training quanta in the total execution quanta. The more *train ratio* an execution has, the lager training overhead it gets. Table 3 shows that cross-run optimization can reduce the training overhead greatly.

6.3 Results on Memory-Intensive Workloads

Memory intensive workloads would result in more training quanta than non-memory-intensive ones. If all applications

TABLE 3
Training Overheads on 8-Core System

Scheme	Train Ratio	Training overhead
Train unlimited	25.7%	6.3%
Train 3%	13.4%	3.0%
Train 2%	10.4%	2.0%
Train 1%	5.7%	1.0%
Cross-Run	2.0%	0.2%

in a workload are memory-intensive, training overhead may become unacceptable, and the accuracy of IPC_{alone} estimation may also be affected. We evaluate the effectiveness of FPS on 5 manually constructed and representative memory-intensive workloads. Evaluation results show FPS can still eliminate system unfairness significantly. When we apply cross-run optimization to memory-intensive workloads, system unfairness is 1.12, which is better than 1.18 under training-unlimited. At the same time, system throughput degradation is only by 1 percent. Training overhead and throughput degradation are greatly reduced. More details are provided in the supplemental materials (Section 4.4), which is available online.

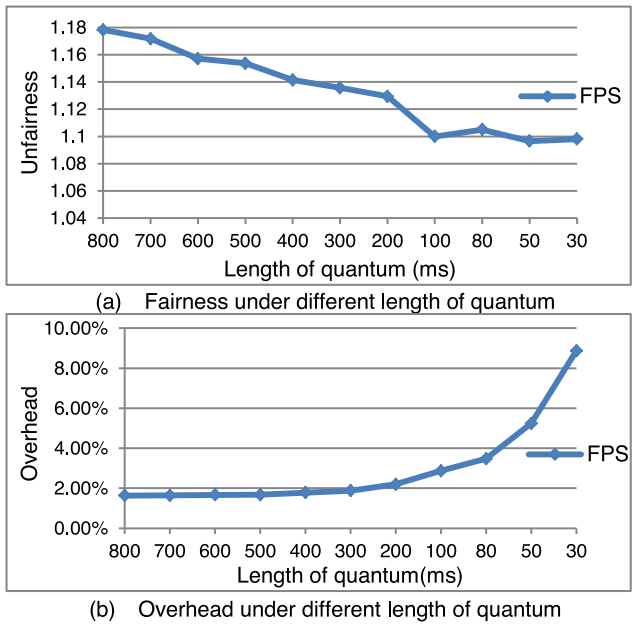


Fig. 6. 4-Core shared-cache-same-chip evaluation results.

6.4 The Length of a Quantum

Choosing a proper length for a quantum is a tradeoff of many considerations. First, we should consider fairness as well as overhead. Second, performance of quanta in the same phase should be as similar as possible. Third, it should be easily adoptable by OS.

We choose 100 ms as the length of a quantum which is the default time slice of Linux native scheduler under round-robin strategy. We compare the fairness and overhead with other length of quantum within the range of Linux time slice. All the evaluation results are on the 4-core shared-cache-same-chip mode.

Fig. 6a shows the average fairness of all benchmarks under different length of quantum. Fig. 6b shows the corresponding overhead. Lager length of quantum has low overhead but it brings poorer fairness benefit. When we decrease the length of quantum toward 100 ms, fairness improves greatly with a slight growth in the overhead. When we set the length below 100 ms, fairness cannot get much better while the overhead increase rapidly.

For each identified phase, we compute the *relative standard deviation* (% RSD) of the IPC_{alone} for the quanta within the phase. Smaller %RSD means the IPCs of quanta that in the same phase are similar, and is better. Finally, we report the application's phase identification accuracy by calculating the weighted %RSD of all phases, as

$$WRSD = \sum_{for\ each\ phase\ p} w_p \times \mathrm{RSD}_p, \quad where$$
$$w_p = \frac{n_p}{\sum_{for\ each\ phase\ i} n_i}, \tag{6}$$

where n_p is the number of quanta that belong to phase p. Each phase is assigned with a weight w_p. Fig. 8 shows the phase accuracy of all benchmarks. The average WRSD is 5 percent, this shows that the phase identification scheme could successfully classify the executed quanta so that the performance in each phase is quite similar.

Therefore, 100 ms is a proper length in our evaluation when we consider all of these.

6.5 Impact of Co-Runners on Cross-Run Optimization

In the previous sections, cross-run optimization is applied to the same workload which means applications run with same co-runners. In this section, we run workloads 20 times. Each time, a randomly selected workload runs. In this way, random co-runners run every time.

Fig. 7 presents the train ratio in these 20 runs. Each vertical bar along the x-axis represents a random workload run. The y-axis shows train ratio in this execution.

During the first execution, the repository is empty, thus the train ratio is very close no matter whether cross-run applied or not. As additional executions occur, the valid phase accumulated. Train ratio under cross-run optimization is reduced dramatically.

6.6 Impact of Input Change on Cross-Run Optimization

There are nine benchmarks with multiple input sets for the reference workload in the SPEC CPU 2006. We extend WL#9 to make it contain the whole multiple input benchmarks.

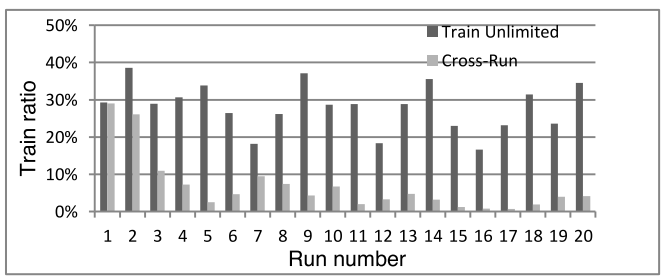


Fig. 7. Train ratio comparison of train unlimited and cross-run on 8-core systems.

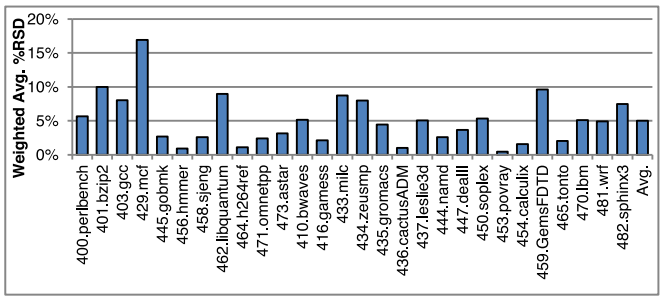


Fig. 8. Phase identification: Weighted relative standard deviation (lower is better).

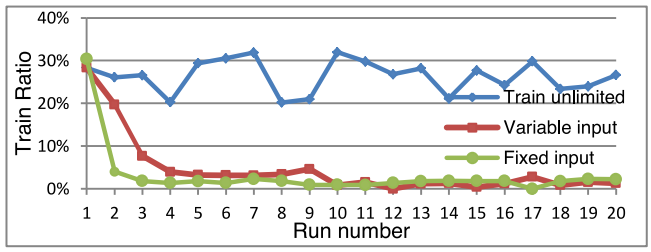


Fig. 9. Train ratio comparison of train unlimited and cross-run when input set changes across the 20 runs.

We run the extended WL#9 20 times and change the inputs of benchmarks in a round-robin manner.

Fig. 9 presents the variation of train ratio when input set changes across the 20 runs. Each point along the x-axis represents a workload run. The y-axis shows train ratio. *Train unlimited* means FPS without cross-run optimization. *Variable input* means we use cross-run optimization with variable inputs while *fixed input* represents cross-run with fixed input. If we don't apply cross-run optimization to FPS, each run is independent. When we use cross-run optimization, each run is a *training run* for the later runs while it is actually a *production run* itself.

The difference between *Train unlimited* and *Variable Input* in a run indicates some phases don't need to be trained under this input because they have already appeared in previous runs. *Fixed Input* gets significant low train ratio since the second run because few new phases are appeared. After running 9 times, train ratio of *Variable Input* is close to that of *Fixed Input* because all the inputs have been executed and valid phases all have been accumulated.

7 Related Work

Techniques that improve fairness on multiprocessors have been widely studied, and contention in memory resources was identified as the primary cause of such unfairness [2], [3], [6], [7], [8], [22]. There are mainly three different approaches to improve fairness [7], i.e., using 1) resource usage (RUM), 2) resource performance (RPM) and 3) overall performance (OPM) as a metric.

Techniques using RUM try to allocate the amount of demanded resources to applications. Antonopoulos et al. [13] designed cache partition techniques to make sure that high-priority applications get more cache space. Sherwood et al. [15] proposed *fair queuing* on the memory controller to ensure that each thread receives its allocated fraction of memory bandwidth. However, providing all applications with same amount of resources does not necessarily produce best performance because the demand for resources is highly application-dependent. Techniques using RPM try to guarantee the applications a certain level of resource performance. Mutlu and Moscibroda designed a memory scheduling policy to let equally-weighted applications have the same increase in memory-related stall time [6]. Fedorova et al. [3] proposed a thread scheduling policy to let the threads have similar miss rates on shared cache. However, other complementary techniques are still needed to bridge the gap between the resource-performance and the observed application-performance (e.g., IPC). By comparison, FPS targets application performance directly using an objective similar to OPM.

For OPM and RPM, the most challenging task is to estimate what the performance would be if an application runs alone while it actually runs concurrently with others. For example, in [6], authors added special hardware counters and triggers in memory controllers to estimate the memory stall time when the application runs alone. Similar hardware support is used in their follow-up work [2], in which the shared-cache contention is also measured. Because of the complicated mechanisms in memory devices and their interactions with the processor pipeline, precise analytical modeling of their performance is very difficult. In this paper, to estimate what an application's performance would be if it runs alone, we used a totally different runtime approach: we make use of the phase behavior of an application and estimate IPC_{alone} directly by constructing a *low-contention* environment for it. Our approach is software based and does not need any special hardware support.

In order to provide system fairness, most prior work manages the shared resources and changes the behavior of applications when they share the resources. In this paper, we use a process scheduling approach to deal with the problem [23]. Although contention-aware thread scheduling policies have been widely studied, most of them focus on system throughput [4], [5], [13], [14]. A fairness-oriented thread scheduling policy was proposed in [3]. It targeted shared cache contention and used RPM as its objective. In [12], authors came up several mechanisms for supporting the notion of priority while still allowing symbiotic job-scheduling [11]. Our approach also can be easily extended to support different priorities by defining *WeightedProgress*. By comparison, FPS mainly targets main memory contention, which has been identified as the most dominated cause for an application's performance degradation [1], and it uses OPM.

We conduct additional comparisons to [24], [25], [26], [27], [28], [29]. Details are presented in the supplemental materials (Section 6), which is available online.

8 Conclusion

We proposed a fair progress scheduling policy (FPS) to provide performance fairness on shared-memory multiprocessors. The basic idea is that, given the same amount of CPU time, if an application did less effective work than others because it suffers bigger slowdown due to resource contention, FPS would allocate extra time quanta to it.

We define the forward progress to quantitatively measure the effective work of an application. The challenge when calculating the progress at runtime is to estimate the run-alone performance in each executed quantum while the application is actually running simultaneously with others. Our solution is to classify the execution quanta of application into phases, and obtain their estimated IPC_{alone} by identifying or constructing the low-contention co-scheduled applications. We then extend the performance information to other quanta that belong to the same phase in order to help estimating their progress.

We also proposed cross-run optimization to alleviate the overhead of online training.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grants 61332009, 61303052, 61303051, the National High Technology Research and Development Program of China (2012AA010901, 20121040), the National Basic Research Program of China (20118034), the Innovation Research Group of NSFC (61221062). Z. Wang is the corresponding author.

REFERENCES

[1] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Shared Resource Contention in Multicore Processors via Scheduling," *Proc. 19th ACM SIGPLAN Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[2] E. Ebrahimi, C.J. Lee, O. Mutlu, and Y.N. Patt, "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory System," *Proc. 19th ACM SIGPLAN Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[3] A. Fedorova, M. Seltzer, and M.D. Smith, "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler," *Proc. 16th ACM SIGARCH Int. Conf. Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[4] D. Xu, C. Wu, and P.C. Yew, "On Mitigating Memory Bandwidth Contention through Bandwidth-Aware Scheduling," *Proc. 19th ACM SIGARCH Int. Conf. Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[5] H.Y. Cheng, C.H. Lin, J. Li, and C.L. Yang, "Memory Latency Reduction via Thread Throttling," *Proc. IEEE /ACM 43th SIGMICRO Int. Symp. Microarchitecture (MICRO)*, 2010.

[6] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," *Proc. IEEE /ACM 40th SIGMICRO Int. Symp. Microarchitecture (MICRO)*, 2007.

[7] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "QoS Policies and Architecture for Cache/Memory in CMP Platforms," *Proc. ACM SIGMETRICS Int. Conf. Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2007.

[8] R. Iyer, "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms," *Proc. 18th ACM Int. Conf. Supercomputing (ICS)*, 2004.

[9] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Throughput and Fairness in SMT Processors," *Proc. IEEE Symp. Performance Anal. Syst. Software (ISPASS)*, 2001.

[10] K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith, "Fair Queuing Memory Systems," *Proc. 39th IEEE /ACM SIGMICRO Int. Symp. Microarchitecture (MICRO)*, 2006.

[11] A. Snavely and D.M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor," *Proc. 9th ACM SIGPLAN Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[12] A. Snavely, D.M. Tullsen, and G. Voelker, "Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor," *Proc. ACM SIGMETRICS Int. Conf. Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2002.

[13] C.D. Antonopoulos, D.S. Nikolopoulos, and T.S. Papa-theodorou, "Scheduling Algorithms with Bus Bandwidth Considerations for SMPs," *Proc. 32th Int. Conf. Parallel Processing (ICPP)*, 2003.

[14] E. Koukis and N. Koziris, "Memory and Network Bandwidth Aware Scheduling of Multiprogrammed Workloads on Clusters of SMPs," *Proc. 12th IEEE Int. Conf. Parallel Distributed Systems (ICPADS)*, 2006.

[15] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *Proc. 10th ACM SIGPLAN Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[16] A.S. Dhodapkar and J.E. Smith, "Comparing Program Phase Detection Techniques," *Proc. 36th IEEE/ACM SIGMICRO Int. Symp. Microarchitecture (MICRO)*, 2003.

[17] T. Sherwood, S. Sair, and B. Calder, "Phase Tracking and Prediction," *Proc. 30th ACM/IEEE Int. Symp. Computer Architecture (ISCA)*, 2003.

[18] T. Sherwood, E. Perelman, and B. Calder, "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications," *Proc. 10th ACM SIGARCH Int. Conf. Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[19] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture," *Proc. IEEE Symp. High-Performance Computer Architecture (HPCA)*, 2005.

[20] C. CaBcaval and D.A. Padua, "Estimating Cache Misses and Locality Using Stack Distances," *Proc. 17th ACM Int. Conf. Supercomputing (ICS)*, 2003.

[21] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," *Proc. 13th ACM SIGARCH Int. Conf. Parallel Architectures Compilation Techniques (PACT)*, 2004.

[22] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," *Proc. ACM /IEEE 35th Int. Symp. Computer Architecture (ISCA)*, 2008.

[23] D. Xu, C. Wu, P.C. Yew, J. Li, and Z. Wang, "Providing Fairness on Shared-Memory Multiprocessors via Process Scheduling," *Proc. 12th Joint ACM SIGMETRICS/PERFORMANCE Int. Conf. Measurement Modeling Comput. Syst. (SIGMETRICS)*, 2012.

[24] H. Hoffmann, J. Eastep, M.D. Santambrogio, J.E. Miller, and A. Agarwal, "Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments," *Proc. IEEE/ACM 7th Int. Conf. Autonomic Comput. Commun. (ICAC)*, 2010.

[25] M. Maggio, H. Hoffmann, M.D. Santambrogio, A. Agarwal, and A. Leva, "Controlling Software Applications via Resource Allocation within the Heartbeats Framework," *Proc. IEEE 49th Int. Conf. Decision Contr. (CDC)*, 2010.

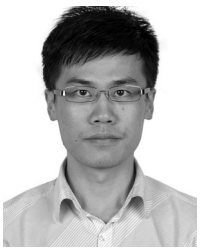
[26] M. Arnold, A. Welc, and V.T. Rajan, "Improving Virtual Machine Performance Using a Cross-Run Profile Repository," *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA)*, 2005.

[27] K. Tian, Y. Jiang, E. Zhang, and X. Shen, "An Input-Centric Paradigm for Program Dynamic Optimizations," *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA)*, 2010.

[28] K. Tian, E. Zhang, and X. Shen, "A Step towards Transparent Integration if Input-Consciousness into Dynamic Program Optimizations," *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA)*, 2011.

[29] J. Mars, L. Tang, R. Hundt, K. Skadron, and M.L. Soffa, "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-Locations," *Proc. 44th IEEE /ACM SIGMICRO Int. Symp. Microarchitecture (MICRO)*, 2011.

Chenggang Wu received the PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences (ICT CAS) and his research was supported by National Science Foundation of China (NSF), the National High Technology Research and Development Program of China, and the National Science and Technology Major Project of China. He is currently an associate professor at the Key Laboratory of Computer System and Architecture of ICT CAS. He serves as the general co-chair of CGO 2013, program co-chair of APPT 2013, and the program committee member of PLDI 2012, PLC 2012, PPPJ 2014, and AMAS-BT. He is serving as the member of Computer Architecture Professional Committee of China Computer Federation. His research interests include the dynamic compilation, including binary translation, dynamic optimization, bug detection on concurrent program, and software security.



Jin Li received the bachelor's degree from Beijing Jiaotong University in 2010. He is currently working toward the PhD degree in the State Key Laboratory of Computer Architecture, Institute of Computing Technology. His research interests include dynamic compilation, binary translation and optimization, Android application analysis.

Di Xu received the bachelor's degree from Harbin Institute of Technology in 2006 and the PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences, in 2012. He is currently working at the IBM Research, China. His research interests include distributed computing, cloud middleware.

Pen-Chung Yew has been a professor in the Department of Computer Science and Engineering, University of Minnesota since 1994, and was the head of the department and the holder of the William-Norris Land-Grant chair professor between 2000 and 2005. He also served as the director of the Institute of Information Science (IIS) at Academia Sinica in Taiwan between 2008 and 2011. Before joining the University of Minnesota, he was an associate director of the Center for Supercomputing Research and Development (CSRD) at the University of Illinois at Urbana-Champaign. From 1991 to 1992, he served as the program director of the Microelectronic Systems Architecture Program in the Division of Microelectronic Information Processing Systems at the National Science Foundation, Washington, DC. He served as the editor-in-chief of the *IEEE Transactions on Parallel and Distributed Systems* between 2000 and 2005. He has also served on the organizing and program committees of many major conferences. His current research interests include system virtualization, compilers and architectural issues related multi-core/many-core systems. He is a fellow of the IEEE.

Jianjun Li received the BS degree in computer science from Harbin Engineering University in 2006, and the PhD degree from Institute of Computing Technology, Chinese Academy of Sciences, in 2012. He is currently an assistant professor in the State Key Laboratory of Computer Architecture, Institute of Computing Technology. His research interests include dynamic program analysis, program optimization, and binary translation.

Zhenjiang Wang received the BS degree in computer science from Tsinghua University in 2005,and the PhD degree from Institute of Computing Technology, Chinese Academy of Sciences, in 2011, and joined the State Key Laboratory of Computer Architecture, where he is currently an assistant professor. His research interests include dynamic compilation, binary translation, and optimization.