Providing Fairness on Shared-Memory Multiprocessors via Process Scheduling

Di Xu[1,2], Chenggang Wu[1]*, Pen-Chung Yew[3,4], Jianjun Li[1], and Zhenjiang Wang[1]

[1]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences

[2]Graduate University of Chinese Academy of Sciences, Beijing, China

[3]Department of Computer Science and Engineering, University of Minnesota at Twin-Cities, Minnesota, USA

[4]Institute of Information Science, Academia Sinica, Taiwan

{xudi, wucg, lijianjun, wangzhenjiang}@ict.ac.cn, yew@cs.umn.edu

ABSTRACT

Competition for shared memory resources on multiprocessors is the most dominant cause for slowing down applications and makes 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. Its strategy is to force the equally-weighted applications to have the same amount of slowdown when they run concurrently. The basic approach is to monitor the progress of all applications at runtime. 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. Our policy also allows different weights to different threads, and provides an effective and robust tuner that allows the OS to freely make tradeoffs between system fairness and higher throughput.

Evaluation results show that FPS can significantly improve system fairness by an average of 53.5% and 65.0% on a 4-core processor with a private cache and a 4-core processor with a shared cache, respectively. The penalty is about 1.1% and 1.6% of the system throughput. For memory-intensive workloads, FPS also improves system fairness by an average of 45.2% and 21.1% on 4-core and 8-core system respectively at the expense of a throughput loss of about 2%.

Categories and Subject Descriptors

D.4.1 **[Operating Systems]**: Process Management – *scheduling*

General Terms

Management, Design, Performance

Keywords

Process Scheduling, Performance Fairness, Memory Bandwidth

1. INTRODUCTION

Shared-memory multi-core processors are the most prevalent platforms used today. When applications run concurrently on such systems, the competition for the 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). Figure 1 shows the effect of resource contention on the performance of four equally-weighted and concurrently running applications, *perl*, *bwaves*, *milc* and *libquantu*m, all from SPEC2006. They run on a 4-core CMP with private cache (details are in Section 4.1). Compared to the isolated run, the execution time of *perl* increases to 1.10X, while the execution time of *libquantum* increases to 1.62X because *libquantum* is memory-intensive and 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 1.19X to 1.47X. It shows that the performance of an application highly depends on its co-runners and can change unpredictably due to resource contention. It violates the assumption of weight-based CPU time allocation policy in the OS, and exacerbates the need for quality of service (QoS) on such systems.

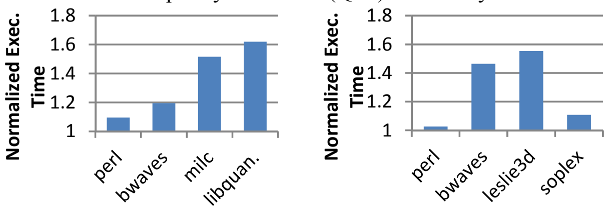


Figure 1. Performance variations of concurrent applications

In order to provide performance fairness to concurrently running applications, some prior works 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 fair performance on demanded resources, such as cache miss rates [3, 20] 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 several prior works [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. The basic mechanism is: at runtime, we use the data gathered from the performance monitoring unit (PMU) and an analytical model to derive the amount of its *forward progress* after the execution of a time quantum. If we find an application suffered more slowdown (thus accumulated less *progress*) than others within the same time quantum, we would 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 greatest challenge is to estimate its performance if it runs alone on the system, while it is actually running simultaneously with others [2,

*To whom correspondence should be addressed

6]. In this paper, we proposed a software-based approach. Firstly, we classify all executed time quanta into phases [14-17]. The performance in the time quanta of the same phase is very similar. In each phase, we estimate its run-alone performance via identifying the *low-contention* time quanta (Section 4.1), in which 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*.

For a phase without such *low-contention* time quanta, we force a time quantum to run with the desired *low-contention* co-runners by turning it into a *training* quantum (Section 4.3). Training improves the accuracy of run-alone performance estimation and the system fairness significantly at the expense of some CPU idleness and system throughput degradation. To mitigate the problem, we proposed techniques that can effectively reduce the training overhead. In addition, by setting an upper-bound on the training overhead, FPS provides the OS with an effective and robust tuner to tradeoff between system fairness and throughput. It makes FPS adaptive to different fairness objectives.

Our experiments are on a commercial server with Intel multicore processors. Evaluations show that compared to the native OS scheduler, FPS could reduce 53.5% and 65.2% of system unfairness on a 4-core private-cache system and a 4-core share-cache system with a throughput degradation of 1.1% and 1.6%, respectively. For memory-intensive workloads, FPS reduces the system unfairness by an average of 45.2% and 21.1% on 4-core and 8-core systems respectively when capping the training overhead to 2%. Even without any training, FPS still gets 15% better fairness than the default OS scheduler, and the change in throughput is quite negligible.

Our contributions are as follow:

- We propose a scheduling policy to provide performance fairness on commodity systems. The policy can significantly improve system fairness at the expense of slightly decreased throughput, and it also enforces thread priorities/weights.
- We propose a practical, phase-based run-time scheme to obtain the run-alone performance of an application while it actually runs simultaneously with others. It is software-based and does not need any special hardware support.
- An effective and robust tuner is provided to let the user freely make tradeoffs between system-fairness and higher throughput.

The rest of this paper is organized as follow: Section 2 introduces the related work. Section 3 shows the overview of our policy. The run-alone-performance estimation scheme is introduced in Section 4. In Section 5, we briefly discuss the system software support. Second 6 and Section 7 introduce the evaluation methodology and results, respectively. Finally, we conclude this paper in Section 8.

2. RELATED WORK

The techniques to provide performance fairness on multiprocessors have been widely studied, and memory resource contention was identified as the primary cause for unfairness [2, 3, 6-8, 21]. There are mainly three different ways to target 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 demanded amount of resources to applications. R. Iyer et al. [12] designed cache partition techniques to make sure that high-priority applications get more cache space. Kyle J. Nesbit et al. [14] proposed fair queuing on the memory controller to ensure that each thread receives its allocated fraction of memory bandwidth. However, providing different applications with the same amount of resources does not necessarily produce fair performance because the demand for resources is highly application-dependent. Techniques using RPM try to guarantee the applications a certain level of resource performance. O. Mutlu and T. Moscibroda designed a memory access scheduling policy to let equally-weighted applications have the same increase in memory-related stall time [6]. A. Fedorova et al. [3] proposed a thread scheduling policy to let the threads achieve execution times if they have the same miss rate on shared cache. However, other complementary techniques are still needed to bridge the gap between the resource-performance and the final application-performance (e.g. IPC). By comparison, FPS targets application performance directly using the OPM objectives.

For using OPM and RPM, the most challenging task is to estimate what the situation *would* be if the application runs alone while it actually runs simultaneously with others. For example, in [6], authors added special hardware counters and triggers in memory controller to estimate what the memory stall time is if the application runs alone. Similar hardware support is used in their follow-up work [2], in which the shared-cache contention is also counted. Because of the complicated working mechanism in memory devices and their interactions with the processor pipeline, precise analytical modeling of performance is still very difficult. In this paper, in order to estimate what the application performance (IPC) would be if it runs alone, we proposed a totally different runtime approach: we make use of the phase behavior of applications and identify its 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 works manage the shared resources and change the behavior of applications when they share the resources. In this paper, we use a process scheduling approach to deal with the problem. Although contention-aware thread scheduling policies have been widely studied, most of them focus on system throughput [4, 5, 12, 13]. A fairness-oriented thread scheduling policy has been proposed in [3]. It targets shared-cache contention and uses RPM as its objective. 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.

3. POLICY OVERVIEW

3.1 Specifying the Fairness Target

Similar to previous works [2, 6, 9, 11], we assume a system is *fair* if equally-weighted applications have the same slowdown when they run concurrently on system. As shown in Equations (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.

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

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

3.2 Basic Ideas

FPS tries to guarantee the equally-weighted applications to 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}) within the same time period. We define the *forward progress* to quantitatively measure the *effective work* that an application has done. Assume that 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:

$$progress = \sum_{q=1}^{Q} \frac{C_{alone}^{q}}{C_{unit}} = \sum_{q=1}^{Q} \left(\frac{I^{q}}{IPC_{alone}^{q}} \times \frac{1}{C_{unit}} \right) \qquad (3)$$

For a time quantum q, C_{alone}^{q} is the number of CPU cycles if the application runs alone. The application's *progress* is the accumulation of C_{alone}^{q} in each executed time quantum (from 1 to Q) normalized to C_{unit}. For example, an application runs simultaneously with others for C_{unit}*2 cycles but suffers 4X 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 calculating *progress*, I^{q} is the number of executed instructions in quantum q. It can be obtained directly from PMU. The main challenge is to estimate IPC_{alone}^{q}. In Section 4, we will describe our proposed method in more detail.

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          $progress += I/IPC_{alone} \times 1/C_{unit}$
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
```

In order to impose equally-weighted applications the same slowdown, FPS tries to let them achieve the same *progress* within the same given time when they run concurrently. Algorithm 1 shows the basic steps 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_{alone}^{q}. We then repeatedly schedule the application with the smallest *progress* on each available CPU core.

4. ESTIMATING RUN-ALONE PERFORMANCE

To estimate the run-alone performance in each executed quantum, we make use of the phase behavior in applications. Firstly, we group executed quanta into phases, and use the attribute that the performance of the quanta in the same phase should be similar [14-17]. If we know IPC_{alone}^{q} 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_{alone}^{q} without running the application alone.

4.1 Identifying *Low-Contention* Applications in a Quantum

Competition for shared memory resource is the primary cause for performance variations [2, 3, 6-8, 20, 21]. 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 $IPC_{alone}^{q} \approx IPC_{shared}^{q}$ in those cases.

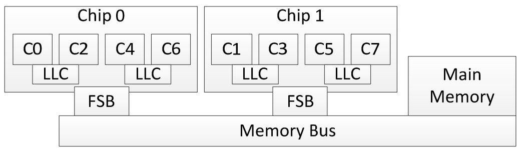


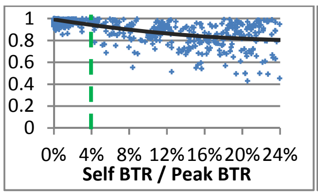
Figure 2. Architecture overview of the evaluation system.

In this section, we use measured results on real systems to better explain the phenomenon. Figure 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 2 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.

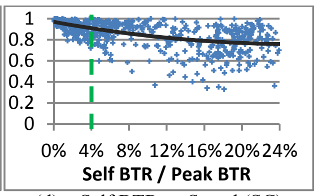
In this paper, we call the execution part of an application during a scheduling time quantum an *application segment*, or *segment* for short. In each segment, its IPC_{shared}^{q} can be obtained via PMU. We also get its real IPC_{alone}^{q} by querying an offline performance profiling file, which is generated by executing the application *alone* on the same system. At last we can calculate its speedup as $IPC_{shared}^{q}/IPC_{alone}^{q}$. Note the method of obtaining IPC_{alone}^{q} 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, Figures 3(a) and 3(b) plot the evaluation results of the same randomly selected and representative 1k segments in private-cache mode. Figure 3(c) shows a subset of them. The three *low-contention* cases are as follows:

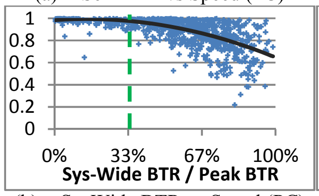
- The bandwidth requirement of a segment is extremely low (Criterion 1). In this case, its performance degradation due



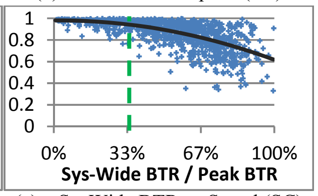
(a) Self BTR vs Speed (PC)



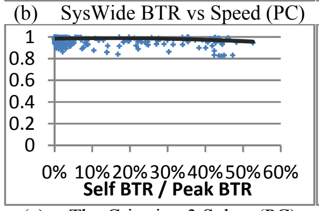
(d) Self BTR vs Speed (SC)



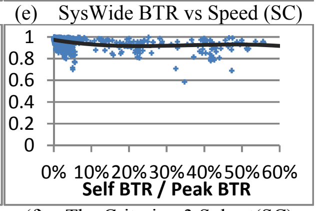
(b) SysWide BTR vs Speed (PC)



(e) SysWide BTR vs Speed (SC)



(c) The Criterion 3 Subset(PC)



(f) The Criterion 3 Subset(SC)

Figure 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.

to bandwidth contention is always small, no matter what applications it runs with. Evaluation results shown in Figure 3(a) 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% 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}.

- 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 bandwidth of bus, 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 can assume IPC^{q}_{shared} is similar to IPC^{q}_{alone} for *all* the concurrent segments.

- 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. Figure 3(c) shows a subset of evaluated segments that belong to this case. Compare it with Figure 3(a), 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, Figures 3(d,e,f) 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 of the application segments 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 abovementioned three criteria would have a 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.

4.2 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 [14-17] phase identification scheme to classify similarly behaved quanta into a phase. BBV analysis has been shown to be an effective method of identifying phases in programs [14, 15].

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 Equation (4), where x_i and y_i is the element of vector *X* and *Y* respectively, and *N* is the number of static basic blocks in application binary. If the Manhattan Distance of two BBVs is smaller than a threshold, the corresponding segments are in the same phase.

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

Evaluations on PMU sampling overhead and phase identification accuracy are in Section 7.2 and 7.7.

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	$IPC_{phase}=$ $(IPC_{phase} * n + IPC^{q}_{shared})/(n+1)$

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*, firstly 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 meets, its performance data is considered *valid*, otherwise, it is *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 *invalid* histories. After the phase gets its first valid execution, the intended value of IPC_{phase} is the average of all valid histories. 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}.

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 could not be larger than that of when it runs alone.

4.3 Training Quantum

During execution, a phase of an application may never have a quantum that fits any of the three *low-contention* cases, and it degrades the accuracy of the estimation. In that case, we can temporarily change the scheduling policy and inject a *training* period to create necessary *low-contention* situations for the phase.

For each application, we use a Markov predictor to predict its phase changes based on its full histories. To tolerate incorrect prediction, the predictor reports several phases that are most likely to appear in next quantum according to their transferring probabilities. If one of the predicted phases has appeared at least twice in history and we have not obtained its valid IPC_{alone} yet, that application will become a training-target-candidate. A training target selection policy will be used to determine the final training target if there are multiple candidates (Section 4.5), then FPS transfers the scheduling policy to *training* in next quantum

Algorithm 2: Scheduling Algorithm - Training

```
01  // schedule the training target app t
02  move app t to run-queue
03  $BTR_{remain} = BTR_{SyswideWeak} - BTR^{t}_{required}$
04  $CORE_{remain} = \#cores$-1
05  // limit system-wide BTR (to meet criterion 2)
06  while $BTR_{remain}$>0 do
07      for each app $p$ in wait-queue do
08          $FITNESS^{P} = \frac{1}{\left|\frac{BTR_{remain}}{CORE_{remain}} - BTR^{P}_{required}\right|}$
09      end for
10      find app with maximal fitness → app f
11      $BTR_{remain} -= BTR^{f}_{required}$
12      if ($BTR_{remain} > 0$) then
13          move app f to run-queue
14          $CORE_{remain} --$
15      end if
16  end while
17  // meet criterion 3
18  if #mem-intensive in run-queue is no larger than 1 then
19      for each app $p$ in wait-queue do
20          if ($BTR^{p}_{required} \leq BTR_{self}$ && $CORE_{remain} > 0$)
21              move app $p$ to run-queue
22              $CORE_{remain} --$
23      end for
24  end if
```

Algorithm 2 shows the training algorithm. Firstly, the training target is selected to run. The policy then selects proper co-runners to let the system-wide bandwidth requirement close to but not exceeding $BTR_{SyswideLow}$ (to meet Criterion 2). To do so, it goes through all un-selected applications to compute their *fitness*, and the application whose bandwidth requirement is closest to the remaining bandwidth per remaining core has the highest priority to be scheduled. Similar fitness calculation has been used in previous works [4, 12, 13] as a throughput-oriented scheduling policy. After that, the policy selects other non-memory-intensive applications, if possible, to meet Criterion 3. Finally, there may be remaining CPU cores, but we cannot assign applications to them because of the bandwidth constraints. In Section 5, we will discuss the impact of such CPU idleness on the system throughput.

Task color: If some applications always be selected as co-runner of the training target, it would upset the desired system fairness. So we improve the training policy with a task-coloring mechanism. At runtime, we classify the applications into 2 exclusive categories (colors). The application with maximal *progress* currently and those whose *progress* would become the maximal if it executes in the next quantum (predicted according to the histories in phase table) are colored as black. Others are non-black. Scheduling a black application may make its *progress* even larger than current maximal *progress* and degrade system fairness. So the improved training policy first finds the best-fit application among the non-black ones (Line 07 of Algorithm 2 is changed to "**for** each non-black app *p* in *wait-queue* **do**"). The black applications will not be scheduled unless the number of non-black applications is smaller than the number of CPU cores.

Reduction of Training Period: For some phases, such as iterative execution of the body of a loop (so called loop-dominated phases), the performance of part of its execution already represent that of the entire run. This phenomenon gives us an opportunity to further reduce the CPU idleness caused by training. In this paper, we set the length of a training quantum to one-quarter of a normal quantum. If the trained phase is loop-dominated, the distribution of the sampled IPs should remain the same for the rest of the quantum, and the partial execution could still get valid IPC_{alone}. If the phase is not loop-dominated, partial execution would result in a very different IP sampling distribution, and the executed quantum would be recognized as different phases. It will not mislead the IPC_{alone} estimation of the targeted phase.

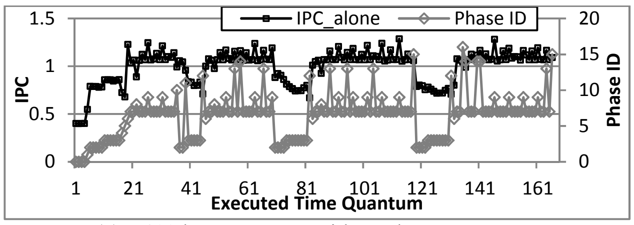
4.4 Training for Applications with Phase Fluctuation

Training by phase prediction requires phases to be correctly predicted. If the accuracy of prediction is low, many training quanta will be wasted, and it also misses the training opportunities for other phases. Figure 4 shows the phase patterns of two different applications that result in different prediction accuracy. Phases of *bwaves* appear in long and stable periods, so the Markov predictor would work well on it. In contrast, the phases of *leslie3d* fluctuate greatly. The reason is its executed code region changes frequently, and when it runs concurrently with others, contention and partial training make the quantum boundary shift unpredictably compared to when it runs alone. As a result, the BBV of each quantum looks different, and forms a hard-to-predict phase sequences.

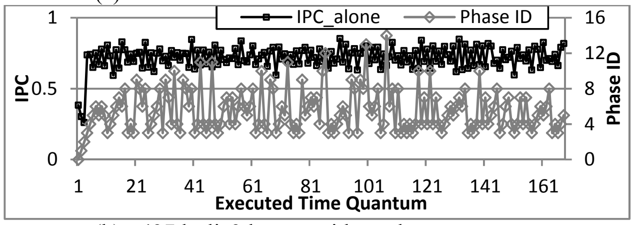
So we introduce another training trigger for applications with fast fluctuating phases. In the beginning, the training is still guided by phase prediction. For an application, if we find the prediction accuracy of quanta is getting very low, the application is identified as having fluctuating phases. It is changed to a sequential training mode: the application is trained continuously in the next N quanta (N=6 in our studies). The motivation is that, if its phases fluctuate, a continuous execution of N quanta is very likely to cover most of the static phases and obtain their valid IPC_{alone}. The scheduling algorithm is unchanged, but the length of training is forced to a full quantum.

4.5 Distribution of Training Quanta Among the Applications

Training could always improve the accuracy of IPC_{alone} estimation of applications. But we found that in a particular case, although the estimation accuracies are improved for some applica-



(a) 410.bwaves – run with random co-runners



(b) 437.leslie3d – run with random co-runners

Figure 4. Applications with different phase behaviors.

tions, it may lead to system fairness degradation because the training quanta are not properly distributed among the applications.

Figure 5 shows such an example. Consider a situation where all the applications in a workload are memory intensive, and all of them will experience large slowdown due to memory contention. The application with maximal and minimal slowdown are B and A respectively, as shown in case 1, and the system unfairness is determined by the difference of their slowdowns. Case 2 shows that when we have sufficient training quanta, the IPC_{alone} estimation accuracies of all applications will improve and approach the upper bound of 100%, and the gap between the estimated maximal and minimal slowdowns becomes smaller and system fairness improves. However, if FPS doesn't have enough training quanta, and they are not properly distributed among the applications, e.g., applications A used most of the trainings to improve its own IPC_{alone} estimation accuracy, while that of the others' remains nearly unchanged, as a result, the gap between the estimated maximal and minimal slowdowns becomes larger, and system fairness degrades (Case 3). What we expect is shown in case 4, where the trainings are properly distributed, so IPC_{alone} estimation accuracies of *all* applications could improve. Intuitively, the improvement of B is larger because it has larger slowdown, and system fairness improves.

In this paper, we propose a training distribution policy. In FPS, an application segment can get its valid estimated IPC^{q}_{alone} from the *low-contention* executions, or from other segments that belong to the same phase. There also exist some segments that have not got their valid IPC^{q}_{alone} estimation. We define the *valid data coverage* of an application to be the ratio between the number of segments that have a valid IPC^{q}_{alone} estimation and the number of all executed time quanta. At runtime, FPS updates the *coverage* metric for each application during its execution. When it is going to insert a *training* quantum, firstly it checks all the applications and finds all the training-target-candidates, and finally, it selects the one with minimal *coverage* among them as the training target. The purpose is to let the applications keep similar *valid data coverage* during execution.

Under this distribution policy, all applications will have chances to use a training quantum, and intuitively the one with larger slowdown will get more training opportunities because it has relatively smaller *valid data coverage*. Note that this policy is always applied in FPS, no matter what the current workload is, we found it especially effective when the workload is memory-intensive and

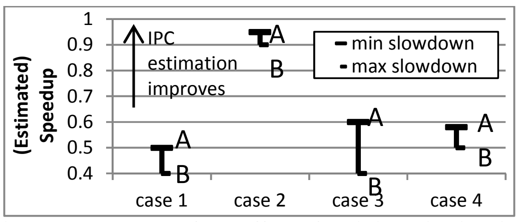


Figure 5. Example of the effect of training distribution

the number of training period is limited, because it could avoid the unbalanced IPC_{alone} estimation accuracy improvement among the applications.

4.6 Shared Cache Considerations

Unlike bandwidth contention, there is no easily identifiable situation in which we can assume the cache contention is small and only depends on simple performance statistics. There may be 2 ways to address the shared cache contention. First, train an application on an unified cache, but this method would involve huge overhead because we need to train at least once for each phase of each application, and at least one CPU core have to be idle. Second, use stack-distance analysis [18-20], but it needs actual memory access traces, which is hard to obtain at runtime without expensive instrumentations.

Nevertheless, we still applied some techniques that could mitigate as much cache contention as possible without introducing further overheads. First, during normal execution in which all CPU cores are used, we adapt the Distributed Intensity Online (DIO) thread-CPU mapping policy [1] to schedule high miss-rate applications to different caches, so that the miss rate is evenly distributed among the caches. Second, when it is in a training period in which at least one CPU core is idle, we will leave the core that shares the same LLC with the training target idle. The remaining applications are assigned to other cores using the DIO policy. Our evaluation shows that, the accuracy of the estimate for IPC_{alone} on a shared-cache system also improves significantly, and system fairness improvement is no less than that on a private cache system.

5. SYSTEM SOFTWARE SUPPORT

A Fairness-Throughput Tuner: Training could improve the accuracy of IPC_{alone} estimation and benefit system fairness, but the side effect is CPU idleness and throughput degradation. We quantify the overhead caused by training using Equation (5):

$$Overhead_{train} = \frac{C_{TrainWaste}}{C_{working}} = \frac{\sum_{for\ each\ training\ quantum\ t}(\#cores - TrainPara^{t}) \times C^{t}_{shared}}{\sum_{for\ each\ app}\sum_{for\ each\ quantum\ q} C^{q}_{shared}} \quad (5)$$

$C_{TrainWaste}$ is the number of cycles wasted in training quanta. It is further calculated as the sum of wasted cycles by each idle CPU core in each training quantum t. $TrainPara^{t}$ is the training parallelism, i.e., the number of simultaneously running applications in training quantum *t*, and C^{t}_{shared} is number of cycles it lasts. $C_{working}$ is the number of cycles that all CPU cores are executing a job. It is the sum of the number of executed cycles in each quantum *q* (denoted as C^{q}_{shared}) of each application.

Recall that in FPS, we reduce the number of training quanta and increase training parallelism by identifying the *low-contention* executions. FPS also uses shorter training quanta. As a result, the overhead of training could be significantly reduced. In addition, by measuring the training overhead and setting a limitation, FPS provides an effective and robust interface to the OS, and it allows

Table 2. Evaluation System Setups

Setup name	CPU cores used	LLC	Bandwidth bottleneck @	Per-core LLC Size (MB)	Per-core BTR (trans./usec)
4core-private-cache (PC)	0, 1, 4, 5	Private	Memory Bus	6	120/4=30
4core-shared-cache-diff-chip (SC-1)	0, 1, 2, 3	Shared	Memory Bus	3	120/4=30
4core-shared-cache-same-chip (SC-2)	0, 2, 4, 6	Shared	Front Side Bus	3	80/4=20
8core	0-7	Shared	Memory Bus	3	120/8=15

tradeoff between system fairness and throughput: the more training we do, the more accurate the estimated run-alone performance becomes, and the more fairness improvement we could get, while the expense is more CPU idleness. It makes FPS adapted to different fairness objectives. Relevant evaluation results will be shown in Section 7.4 and 7.5.

Thread Weight: The policy we have described so far assumes that all applications are of the same priority. The policy can be easily extended to support different thread weights and differentiated performance. We assume a higher weight will result in a higher performance. We define $WeightedProgress = Progress/Weight$. The policy uses *weighted progress* instead of measured *progress*, so an application with a higher weight would appear to achieve less progress than it really does. As a result, it will obtain more time quanta and result in a higher performance.

Task Submission and Completion: Real applications are different in execution time, and they may enter or exit from the system at any time. As a result, not all parts of applications keep running concurrently, so in this situation, FPS does not try to guarantee a fair slowdown of the *entire* run of each application respectively. Instead, we divide the execution into multiple *concurrent regions*. *Concurrent regions* transfer from one to another when the composition of concurrent applications changes, i.e., when a new application start to run on the system, or an old application terminates. The idea of FPS is to guarantee the fairness *within* each *concurrent region*. When a new application is submitted, its *progress* should be initialized to zero. At the same time, for each old application *i* that is still running on the system, its *progress* should be recalculated as $progress^i = progress^i - progress_{min}$, where $progress_{min}$ is the smallest *progress* among all the old applications. The old applications get reduced *progress* to avoid being throttled. When an application exits from the system, the *progress* of other applications remain unchanged.

6. METHODOLOGY

Scheduler Implementation: In order to evaluate the effectiveness of our scheduling policy, we implemented a user-level process scheduler on Linux. The scheduler itself executes as a daemon. When a job is submitted, the scheduler forks out the process and creates a PMU sampling context for it. The scheduler sets a real-time timer to count the scheduling time quanta. When the timer expires, the scheduler is notified by a signal and it enters the scheduling routine. The main steps have been described in Section 3.2. The applications are paused by PTRACE_ATTACH, so that they will not be scheduled by the Linux kernel. At the end, we let the selected applications continue to run using PTRACE_DETACH, reset the quantum timer and let the scheduler sleep to wait for the next notification.

System Setup: We use the same system described in Section 4.1. Table 2 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. When all used cores are on the same chip, the shared front-side bus becomes the bottleneck because its realistic peak bandwidth (80 trans./usec) is smaller than that of the memory bus (120 trans./usec). When the cores are on different chips, the potential pressure from the 2 FSB's to the memory bus is 160 trans./usec. It is larger than the realistic peak bandwidth of the memory bus, so the memory bus becomes the bottleneck. Unless stated otherwise, Table 3 shows the parameters used in our evaluation.

Table 3. 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%

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

In reality, the execution times of applications are different, as we discussed in Section 5, FPS divides the execution into multiple *concurrent regions*, and guarantee the fairness *within* each *region*. The *progress* of applications are recalculated when *region* changes, so the scheduling results of *regions* are relatively independent of each other, and we can focus on the results of some specific and representative *regions* that created by length normalization.

It is more challenging to evaluate the first 10 seconds than in a longer run because the number of quanta in each phase becomes relatively smaller, so the workloads' throughput becomes more vulnerable to the CPU idleness due to training. *483.Xalancbmk* is excluded from the benchmark pool because it has a long initialization period so its behavior in the first 10 seconds changes monotonically and does not have a repeated phase. Our current implementation does not handle this situation, but it can be extended in the future to detect such an application: it counts the times that each phase appears, and if most of the phase does not repeat, we would let it run under a regular scheduling policy.

To evaluate the generality of FPS, we use 10 randomly generated multiprogramming workloads, as shown in Table 4. There are 8 to 16 randomly selected benchmarks in each workload for our 4-core experiments. We use the metric *Ideal Average Bandwidth Requirement* (called IABW for short) defined in [4] to characterize the average bandwidth requirement of a workload, and the workloads are reordered by their IABW in ascending order. When evaluated on the 8-core system, we use the same benchmark combination but each benchmark is spawned twice. We also use other manually constructed memory-intensive workloads for evaluation purposes. The details are in Section 7.5.

Table 4. Random SPEC workloads. The benchmarks in a workload are listed according their run-alone-BTR from high to low. The expression appx(n) means there are n parallel instance of appx in the workload.

WL index	IABW	Composition
WL#05	47.1	leslie(2) zeusmp(1) soplex(1) wrf(1) gobmk(1) perl(1) hmmer(2) namd(2) gamess(1) tonto(1)
WL#02	49.3	libquantum(1) leslie(1) lbm(1) sphinx3(1) soplex(1) wrf(2) sjeng(2) gobmk(1) gromacs(1) dealII(1) h264ref(1) gamess(2) tonto(1)
WL#01	49.9	milc(1) lbm(1) cactusADM(1) astar(1) omnetpp(1) hmmer(1) dealII(1) namd(1)
WL#08	60.0	leslie(1) lbm(2) sphinx3(2) soplex(1) mcf(1) gcc(1) omnetpp(1) wrf(1) gobmk(1) perl(2) dealII(1) gamess(1)
WL#10	65.0	GemsFDTD(1) lbm(1) zeusmp(1) soplex(1) astar(1) calculix(1) gobmk(1) povary(1)
WL#07	65.4	leslie(1) GemsFDTD(1) milc(1) sjeng(1) calculix(2) dealII(2) povary(1)
WL#09	67.4	libquantum(1) GemsFDTD(2) zeusmp(1) gcc(1) astar(1) bzip2(1) perl(1) hmmer(2) dealII(1) h264ref(1)
WL#06	76.6	libquantum(1) milc(1) lbm(2) zeusmp(1) mcf(1) cactusADM(1) astar(1) omnetpp(2) calculix(1) h264ref(1)
WL#04	78.5	libquantum(1) bwaves(1) GemsFDTD(1) sphinx3(1) zeusmp(1) soplex(2) gcc(1) astar(1) hmmer(1) gromacs(1) namd(1) h264ref(1)
WL#03	90.6	libquantum(1) leslie(1) bwaves(1) milc(2) zeusmp(3) mcf(2) cactusADM(1) wrf(2) sjeng(1) perl(1) hmmer(1)

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 3.1. We use 2 metrics to evaluate the impact of FPS to system throughput: *workload turnaround time*, i.e., the time from all applications start at the same time to the last application finishes, and *extended weighted speedup (EWSpeedup)*. The original *weighted speedup* [11] was commonly 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 extended weighted speedup as shown in equation (6).

$$EWSpeedup = \sum_{i=0}^{N-1} \frac{C_{alone}^{i}}{C_{shared}^{i} + C_{wasted}^{i}} \tag{6}$$

The key to the definition is that, when calculating the speedup of applications i, if any CPU core is idle before the number of remaining applications becomes smaller than the number of available cores, the wasted cycles on the idle CPU core are also counted as its execution time. In our policy, CPU idles are caused by the training quanta, and we attribute the wasted cycles to the training target in the corresponding training quantum.

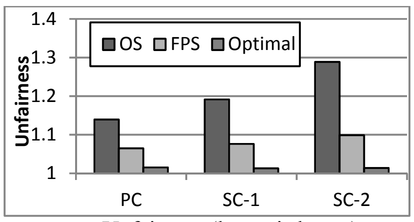
Prior works use Harmonic Mean of Speedups [13] to give a combined measure for both fairness and throughput. We don't use this metric because it is determined by applications' individual IPCs, while our process scheduling technique is not to give the applications fair IPC during execution, but to adjust the CPU time distribution to achieve fair execution time.

7. EVALUATION RESULTS

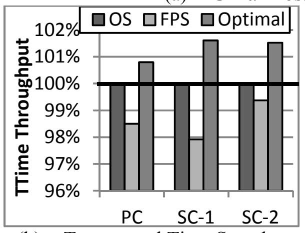
7.1 4-core System Results

Figure 6 shows the average performance of 10 random workloads on the 4-core systems. We evaluate 3 different scheduling policies: the native OS, FPS and Optimal. The optimal policy uses our fairness-oriented scheduling algorithm but $\text{IPC}_{\text{alone}}$ of each quantum comes from offline profiling runs instead of runtime estimation, as described in Section 4.1.

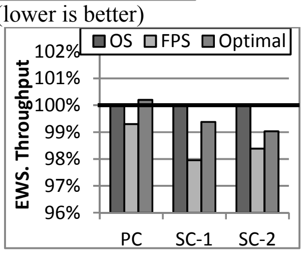
In the private-cache mode, the unfairness under OS scheduler ranges from 1.08 to 1.23, the average is 1.14. FPS improves fairness on all workloads. The average of unfairness is decreased to 1.06, and about 53.5% of the unfairness is eliminated. On the shared-cache-diff-chip mode, the system peak bandwidth 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% 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% of the unfairness. Results show that FPS is effective in eliminating unfairness on both private- and shared-cache systems.



(a) Unfairness (lower is better)



(b) Turnaround Time Speedup over OS (higher is better)



(c) EWSpeedup over OS (higher is better)

Figure 6. Average performance on 4-core systems

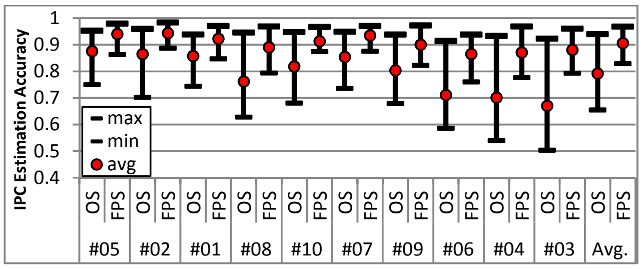
For all the evaluated modes, the most severe system throughput degradation is about 2% compared to OS, no matter whether it was measured in workload turnaround time or EWSpeedup. The average decrease is 1.1%, 2.06% and 1.12% respectively for the three execution modes. The main sources of the degradation is PMU sampling and CPU idleness caused by training.

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.

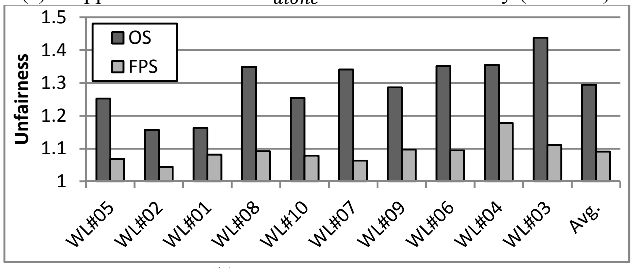
7.2 IPC_{alone} Estimation Accuracy

To give an insight into why FPS improves system fairness, we analyze the accuracy of our IPC_{alone} estimation scheme in this section. We use the evaluation results on the shared-cache-same-chip mode because the resource contention is the most severe among all 4-core modes.

Application Level: For a given benchmark, estimation on each of its quanta has a cumulative effect on the IPC_{alone} estimation accuracy of the entire execution, which is evaluated using Equations



(a) Application-level IPC_{alone} estimation accuracy (1 is best)



(b) System unfairness

Figure 7. 4-core shared-cache-same-chip evaluation results

(7, 8). If we always use ICP_{shared} as IPC_{alone} in our policy, all applications will appear to make the same progress in all quanta. Hence, FPS will allocate the same number of quanta to applications, which has the same effect as using the native OS scheduler. We compare our estimation accuracy to that in this situation.

$$IPC_{esti\_alone} = \frac{\sum_{q=1}^{Q} I^q}{\sum_{q=1}^{Q}\left(I^q / IPC^q_{esti\_alone}\right)} \quad (7)$$

$$EstiAcc = \frac{IPC_{esti\_alone}}{IPC_{alone}} \quad (8) \qquad EstiAcc_{OS} = \frac{IPC_{shared}}{IPC_{alone}} \quad (9)$$

Figure 7(a) reports the maximal, minimal and average estimation accuracy of the benchmarks in each workload respectively. Figure 7(b) report the corresponding system unfairness. As the bandwidth requirement of workloads increase, the slowdowns of applications also increase significantly, e.g., *libquantum* inWL#03 suffers a slowdown of 1.99X, while that of others may be no more than 1.1X. The huge gap between the maximal and minimal slowdown in a workload illustrates the cause of unfairness. The average accuracy of our IPC_{alone} estimation scheme is about 90% for all workloads, and the gap between maximal and minimal is reduced. As a result, system fairness is improved when we further allocate proper fraction of time quanta according to the estimation.

Quantum Level: There are two necessary requirements to achieve high estimation accuracy in each quantum. First, the IPC_{shared} of an application in the low-contention co-schedule and its IPC_{alone} should be as close as possible. We have discussed and evaluated it in Section 4.1. Second, performance of quanta in the same phase should be as similar as possible. The accuracy of phase identification is application-dependent. To evaluate, we run each benchmark for 10 seconds under our phase identification scheme. 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 shown in Equation (10).

$$WRSD = \sum_{for\ each\ phase\ p} w_p \times RSD_p\, , where\ w_p = \frac{n_p}{\sum_{for\ each\ phase\ i} n_i} \quad (10)$$

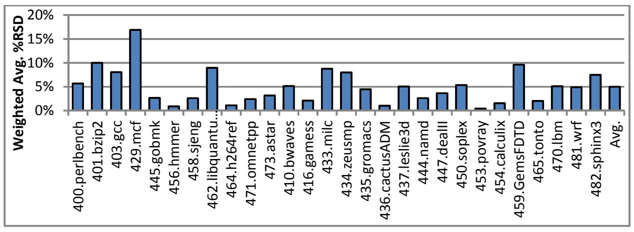


Figure 8. Phase Identification: Weighted Relative Standard Deviation (lower is better)

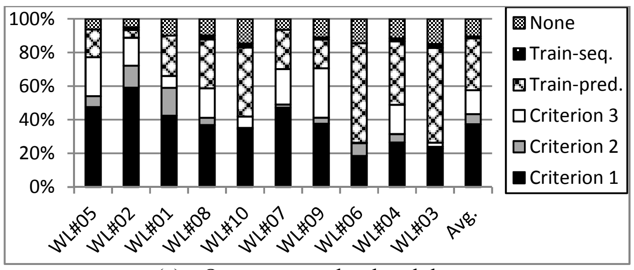
n_p is the number of quanta that belong to phase p. Each phase is assigned with a weight w_p. Figure 8 shows the phase accuracy of all benchmarks. The average *WRSD* is 5%, which shows that the phase identification scheme could successfully classify the executed quanta so that the performance in each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

7.3 Effectiveness of Training

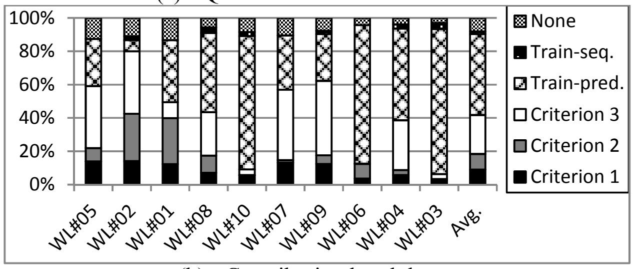
A segment could get its *valid* IPC_{alone} from one of the following sources: the three *low-contention* executions during normal quantum, or training trigged by phase prediction and sequential training. For each application, we compute the distribution of all valid IPC_{alone} sources, and report the average of all applications in a workload, as shown in Figure 9(a). As the bandwidth requirement of the workload increases, the number of *low-contention* execution in normal quanta is reduced. On average, the *valid data coverage* is 90%. About 30% of the quanta get its valid IPC_{alone} via training. 60% are from spontaneous low-contention executions.

However, the contribution to the IPC_{alone} estimation accuracy is different. We evaluate the contribution of each data source in an application as follow: the total IPC_{alone} estimation improvement is $IPC_{esti\_alone} - IPC_{shared}$. To calculate the contribution of source *x*, we find all quanta that use source *x* to estimate IPC^q_{alone}, and replace their $IPC^q_{esti\_alone}$ with IPC^q_{shared}. The purpose is to simulate the situation in which if we did not get a valid IPC_{alone} for this quantum, and have to use IPC_{shared} instead. We recalculate the estimated IPC_{alone} of the entire application in this case as $IPC^{esti\_without\_x}_{alone}$, and the contribution of x is calculated as $\frac{IPC_{esti\_alone} - IPC^{esti\_without\_x}_{alone}}{IPC_{esti\_alone} - IPC_{shared}}$. Figure 9(b) shows the contribution breakdown. It shows that, for most workloads, training plays an important role on improving the accuracy of IPC_{alone}. It is because training deals with those quanta that have inherently high memory intensity, so they have a higher potential to suffer a bigger slowdown that could cause larger system unfairness.

The average contribution of sequential training appears small because it only deals with applications with fluctuating phases, such as *leslie3d, milc* and *zeusmp* in SPEC suite. However, it is important to improve their *coverage* and the IPC_{alone} estimation accuracy. For example, sequential training contributes 28.8% of the estimation accuracy improvement for *leslie3d* in WL#03 and it improves the estimation *coverage* from 88.2% to 93.7% and improves the estimation accuracy from 86.8% to 90.9%, compared to the situation when sequential training is turned off. Significant estimation improvements are also seen in *leslie3d* in WL#02, *milc* in WL#04 and *GemssFDTD* in WL#10, etc. Sequential training can also reduce the number of training quanta because phase prediction accuracy is low for those applications. Sequential training is triggered in 9 applications in the 10 workloads, and on average, it improves the IPC_{alone} estimation accuracy by 2.1% and reduce the number of training quanta by 11%.



(a) Quantum number breakdown



(b) Contribution breakdown

Figure 9. Breakdown of valid IPC^q_{shared} source

Note phases that never get a valid IPC_{alone} also contribute slightly, recall their estimated IPC_{phase} is the average of all invalid histories (refer to Table 1), so it still improves the estimation accuracy for the quanta whose IPC^q_{shared} is smaller than IPC_{phase}.

Table 5 shows the statistics of training on a 4-core system. *Avg. Training Parallelism* is the average number of co-scheduled applications in all the training quanta. Memory intensive workloads suffer more overhead than less memory-intensive ones because there are less spontaneous *low-contention* quanta in the former, so they require more training. WL#03 is the most memory intensive one among the random workloads, about 25.9% of its executed quanta are training, but we can finally reduce the overhead to 4.5% because we reduce the length of a training quantum and increase the training parallelism as much as possible. The average of overhead of all workloads is as low as 1.6%.

Table 5. Training Overheads

WL	#Exec. Quantum	#Train-Pred.	#Train-Seq.	Avg. Training Parallelism	Overhead
#05	376	21	0	2.8	0.5%
#02	472	14	6	3.8	0.1%
#01	246	19	0	2.7	0.8%
#08	498	32	6	2.6	0.9%
#10	266	36	6	2.4	2.4%
#07	274	21	0	3.0	0.6%
#09	413	68	24	3.0	1.8%
#06	468	79	0	2.3	2.3%
#04	504	73	12	2.7	2.3%
#03	715	167	18	2.3	4.5%
Avg.	423.2	53	7.2	2.8	1.6%

7.4 8-core system results

Figure 10 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% 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% of system unfairness at the expense of 4.5% 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 effective and robust.

7.5 Results on Memory-Intensive Workloads

Memory intensive workloads would result in more training quanta than non-memory-intensive ones. If all applications in a workload are memory-intensive, training overhead may become unacceptable, and the accuracy of IPC_{alone} estimation may also be affected. In this section, we evaluate the effectiveness of FPS on 5 manually constructed and representative memory-intensive workloads. Firstly we use a workload that only includes the top 5 applications with the largest BTR in SPEC suite, i.e., *libquantum, leslie3d, bwaves, GemsFDTD* and *milc*. There are 2 concurrent instances for each application. As a result, the IABW of the workload is as high as 196 trans./usec. This workload is denoted as WL-M. And then, we pick up another 4 less-memory-intensive benchmarks: *lbm, mcf, astar* and *gamess*. Their average BTRs vary from 27 trans./usec to nearly zero. We add one of them to WL-M respectively to form another 4 workloads. We run the 5 workloads on 4 system setups. The evaluation results are shown in Table 6. By comparing the results of the workloads and the results on different system setups, we have the following interesting observations.

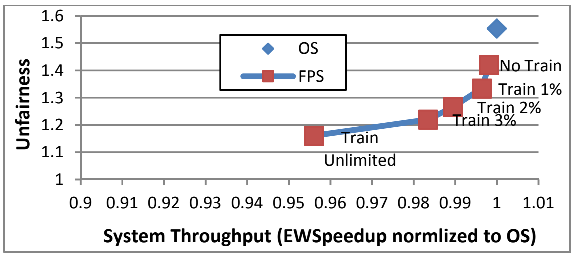


Figure 10. Effect of different training limitation

System unfairness under OS: For an extremely memory-intensive workload where all concurrent applications are memory-intensive, although all of the applications would experience large slowdown, the system unfairness remains relatively small because the difference among the applications' slowdowns is small. For example, When WL-M runs under 4-core-private-cache mode, the maximal and minimal slowdown is 1.92X (for *libquantum*) and 1.65X (for *milc*) respectively, and the system unfairness under OS is only 1.13. Similar results are observed on other 3 system setups, in which the unfairness are 1.12, 1.17 and 1.25, respectively. They are all below the corresponding average of the random workloads. We conclude that severe unfairness is likely to exist when there are both memory-intensive and non-memory-intensive applications. Evaluation results confirmed that when we gradually add a less memory-intensive application into WL-M, the less-memory-intensive application would suffer much smaller slowdown than the memory-intensive ones, and the system unfairness become larger. In the extreme case when WL-M+*gamess* runs in 8-core mode, the unfairness is as high as 4.08, which is the largest among all the workloads that we have evaluated throughout the paper.

Effect of FPS: When the workloads run under FPS and training is not limited, FPS could still effectively eliminate system unfairness even for WL-M where the fairness improvement potential is already small. The average unfairness elimination on the 4 system setups are 59.7%, 66.0%, 61.4% and 71.1% respectively. But the

Table 6. Evaluation results of memory-intensive workloads (positive/negative values are compared to the OS)

System Setup	Workload	OS Unfair-ness	FPS-train-unlimited					FPS-train-limited-2%				
			Unfairness	Training Overhead	T.Time Speedup	EW-Speedup	Unfairness-No Training Distribution	Unfairness	Training Overhead	T.Time Speedup	EW-Speedup	Unfairness-No Training Distribution
PC	WL-M	1.13	1.06 (-52.7%)	17.61%	-11.93%	-10.64%	1.08 (-43.3%)	1.10 (-25.9%)	1.99%	-2.82%	-1.79%	1.47 (+246.9%)
	WL-M+lbm	1.21	1.14 (-34.5%)	20.77%	-14.58%	-13.20%	1.16 (-22.2%)	1.16 (-24.7%)	2.10%	-2.64%	-2.14%	1.44 (+107.0%)
	WL-M+mcf	1.28	1.13 (-53.3%)	15.76%	-10.94%	-9.60%	1.18 (-35.9%)	1.20 (-30.1%)	1.96%	-1.98%	-1.31%	1.34 (+19.9%)
	WL-M+astar	1.44	1.08 (-80.9%)	19.52%	-14.18%	-11.02%	1.12 (-72.9%)	1.25 (-41.9%)	1.97%	-2.85%	-1.86%	1.41 (-6.8%)
	WL-M+gamess	1.65	1.15 (-76.9%)	15.25%	-13.32%	-10.53%	1.16 (-75.2%)	1.28 (-56.5%)	2.02%	-3.10%	-1.83%	1.41 (-37.0%)
	Average	1.34	1.11 (-59.7%)	17.78%	-12.99%	-11.00%	1.14 (-49.9%)	1.20 (-35.8%)	2.01%	-2.68%	-1.79%	1.41 (+66.0%)
SC-1	WL-M	1.12	1.11 (-13.3%)	13.6%	-9.56%	-10.97%	1.11 (-10.3%)	1.10 (-19.1%)	1.98%	-1.58%	-2.49%	1.60 (+383.9%)
	WL-M+lbm	1.32	1.08 (-74.7%)	18.9%	-14.44%	-15.94%	1.07 (-77.4%)	1.19 (-40.7%)	1.97%	-3.91%	-3.90%	1.44 (+36.1%)
	WL-M+mcf	1.31	1.08 (-74.5%)	14.4%	-9.28%	-10.76%	1.08 (-72.5%)	1.12 (-60.8%)	2.02%	-1.43%	-2.50%	1.50 (+61.6%)
	WL-M+astar	1.63	1.10 (-83.8%)	17.7%	-12.01%	-11.10%	1.14 (-78.1%)	1.28 (-56.4%)	2.11%	-2.23%	-2.08%	1.38 (-39.5%)
	WL-M+gamess	1.69	1.11 (-83.8%)	15.3%	-12.13%	-11.21%	1.12 (-83.0%)	1.39 (-43.3%)	2.00%	-2.71%	-3.03%	1.44 (-36.0%)
	Average	1.41	1.10 (-66.0%)	15.98%	-11.48%	-12.00%	1.10 (-64.3%)	1.22 (-44.0%)	2.02%	-2.37%	-2.80%	1.47 (+81.2%)
SC-2	WL-M	1.17	1.15 (-13.4%)	10.60%	-6.02%	-6.95%	1.19 (+9.1%)	1.13 (-22.5%)	1.98%	-1.90%	-2.68%	1.53 (+208.6%)
	WL-M+lbm	1.36	1.15 (-57.3%)	9.45%	-5.46%	-5.17%	1.16 (-54.1%)	1.15 (-56.8%)	1.99%	-1.98%	-1.56%	1.37 (+4.2%)
	WL-M+mcf	1.37	1.17 (-53.7%)	6.92%	-4.39%	-4.41%	1.18 (-51.6%)	1.10 (-74.3%)	2.00%	-1.40%	-1.64%	1.55 (+48.2%)
	WL-M+astar	1.86	1.06 (-92.5%)	9.11%	-5.50%	-4.72%	1.18 (-79.0%)	1.22 (-75.1%)	1.97%	-2.09%	-1.97%	1.32 (-63.5%)
	WL-M+gamess	2.46	1.15 (-90.0%)	9.89%	-6.30%	-5.37%	1.20 (-86.4%)	1.73 (-50.2%)	2.01%	-1.93%	-2.39%	1.40 (-72.4%)
	Average	1.64	1.14 (-61.4%)	9.19%	-5.53%	-5.32%	1.18 (-67.8%)	1.27 (-55.8%)	1.99%	-1.86%	-2.05%	1.43 (+25.0%)
8-core	WL-M	1.25	1.14 (-42.5%)	15.58%	-8.75%	-10.19%	1.17 (-30.8%)	1.19 (-22.6%)	1.80%	-1.84%	-0.68%	1.29 (+15.8%)
	WL-M+lbm	1.51	1.14 (-73.8%)	13.27%	-8.08%	-8.41%	1.14 (-73.0%)	1.49 (-4.1%)	1.77%	-1.39%	-0.51%	1.48 (-7.2%)
	WL-M+mcf	1.29	1.12 (-59.4%)	32.5%	-15.75%	-13.42%	1.17 (-42.1%)	1.24 (-14.9%)	1.93%	-2.57%	-1.19%	1.23 (-18.1%)
	WL-M+astar	2.57	1.13 (-91.6%)	24.2%	-11.54%	-12.08%	1.12 (-92.1%)	2.15 (-26.8%)	1.87%	-0.86%	-2.57%	2.17 (-25.7%)
	WL-M+gamess	4.08	1.36 (-88.2%)	11.0%	-7.32%	-7.15%	1.30 (-90.1%)	2.94 (-37.0%)	1.78%	-2.25%	-2.39%	2.98 (-35.6%)
	Average	2.14	1.18 (-71.1%)	19.31%	-10.29%	-10.25%	1.18 (-65.6%)	1.80 (-21.1%)	1.83%	-1.78%	-1.47%	1.83 (-14.2%)

training overhead is much larger than those random workloads because there are much less spontaneous *low-contention* executions. For example, when WL-M executes in SC-2 mode, the average *valid data coverage* of all applications is only 2% if training is not allowed. If training is not limited, the overheads are between 10% and 20% for the workloads, and the system throughput degradations are between 10% and 15% (as shown in the "T.Time Speedup" and "EWSpeedup" columns). However, we can always use the proposed fairness-throughput tuner to limit the training overhead. It is especially suitable when the unfairness under OS is already relatively small but training is potentially very expensive. For example, FPS could eliminate system unfairness by an average of 35.8%, 44.1%, 55.8% and 21.1% respectively for the 4 system setups when training overhead limitation is set to 2%.

Effect of training distribution: Table 6 also shows the unfairness achieved by FPS when we turn off the proposed training distribution method, which is shown in the "Unfairness-No-Training-Distribution" column. Instead of keeping similar *valid data coverage* of the applications, the compared method always selects the first training candidate that we meet in the run-queue as the final training target. Evaluation results show that, when training is not limited, there is not much difference no matter training distribution is turned on or off, because when there are sufficient training quanta to use, the *valid data coverage* of applications would all approach their upper bound respectively. While the training overhead is limited, different quanta distribution methods would have significant influence on system fairness. We take WL-M running on SC-2 mode as an example. Under OS, the maximal and minimal IPC slowdowns are 3.20X (for *libquantum*) and 2.15X (for *milc*) respectively. When training distribution is off, the unbalanced training resulted in vastly differing *valid data coverage* from 2% (for *GemsFDTD*) to 93% (for *libquantum*). As a result, their estimated IPC slowdowns becomes 2.39X and 1.08X respectively, and the unfairness increases from 1.17 to 1.53. By comparison, when training distribution is on, the maximal and minimal *valid data coverage* is 77% and 53% respectively, and it makes the maximal and minimal estimated IPC slowdowns more similar, which is 1.52X and 1.80X respectively, and system unfairness decreased to 1.13.

7.6 Thread Weight Support

To evaluate the effects of threads with different weights (i.e. priorities), we select 4 different benchmarks whose bandwidth requirements vary greatly from nearly 0% to 46% of the peak, and run 2 concurrent instances for each benchmark and evaluate them on a 4-core shared-cache-diff-chip mode. Figure 11 shows the results. The first set of bars show the performances of benchmarks under the native OS with the same weight. It shows the performance variation due to resource contention. When assigning different weights of 1, 1, 2 and 4 respectively to the applications, OS would allocate proportional number of quanta to application with a higher weight (weighted round robin). The results are shown in the second set of bars. *CactusADM* still runs 21% slower than *gamess* even though they have the same weight, and *bwaves*'s performance is still 1.5% smaller than *gamess* even its number of executed time quanta is doubled compared to the latter. The third set of bars shows the results of FPS. FPS enforces thread weight better because the applications with the same weight have quite similar performance, and the performances of other applications are proportional to their weights.

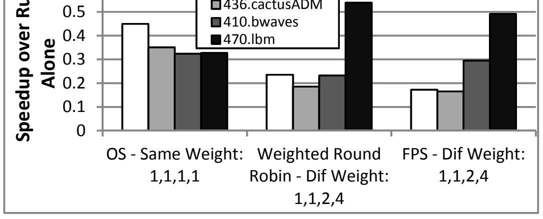


Figure 11. Evaluation of thread weight support

7.7 Overhead of the Scheduler

Compared to the runs under OS, FPS involves overhead mainly from the PMU sampling (interrupt handler) and the scheduling routine (includes the process of PMU data and scheduling). To evaluate the overhead, we run 4 identical applications simultaneously on 4 cores. Compared to OS, execution time under FPS increases 2.87% in total. The overhead from the scheduling routine is 1.68%, and the main source of this part is from PMU and phase related work (overhead from the scheduling algorithm itself is less than 2‰.). Our current user-level scheduler is single-threaded, but PMU data processing for each running application can be naturally distributed to the core on which it executed. Hence, this part of the overhead can be further reduced and made scalable. The remaining 1.19% overhead comes from the PMU sampling, which is distributed among applications, so it is also scalable when the number of cores increases.

8. CONCLUSIONS AND FUTURE WORK

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.

To monitor system unfairness, 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 constructing and identifying 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*.

FPS does not need any special hardware support. Evaluation results show that the IPC_{alone} estimation accuracy is high and it can significantly improve system fairness at the expense of slightly decreased throughput. FPS supports different thread weights. It also provides an effective tuner to let OS freely tradeoff system fairness and higher throughput. Combined with the training distribution policy, it is especially suitable for the memory-intensive workloads where the fairness improvement is smaller but the potential training is expensive.

As a future work, we plan to seek a low-overhead solution to address the issue of cache contention and further improve the estimation accuracy. We also plan to design proper metrics and methodology to address the I/O contention issues.

9. ACKNOWLEDGEMENTS

This research is supported by the Innovation Research Group of NSFC under the grant 60921002, the National Basic Research Program of China (973 program) under the grant 2011CB302504, the National High Technology Research and Development Program of China (863 program) under the grant 2007AA01Z110, 2012AA010901 and 2012AA010902, the National Science and Technology Major Project of China under the grant 2009ZX01036-001-002, and by the U.S. National Science Foundation under the grant CNS-0834599 and CCF-0708822, the Taiwan National Science Council under the grant 100-2220-E-001-001 and 99-2221-E-001-004.

10. REFERENCES

[1] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In ASPLOS-19, 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 systems. In ASPLOS-19, 2010.

[3] A. Fedorova, M. Seltzer, and M. D. Smith. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In PACT-16, 2007.

[4] D. Xu, C. Wu, and P. C. Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In PACT-19, 2010.

[5] H. Y. Cheng, C. H. Lin, J. Li, and C. L. Yang. Memory Latency Reduction via Thread Throttling. In MICRO-43, 2010.

[6] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In MICRO-40, 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. In SIGMETRICS, 2007.

[8] R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In ICS-18, 2004.

[9] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In ISPASS, 2001.

[10] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In MICRO-39, 2006.

[11] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreaded processor. In ASPLOS-9, 2000.

[12] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for smps. In ICPP, 2003.

[13] E. Koukis and N. Koziris. Memory and network bandwidth aware scheduling of multiprogrammed workloads on clusters of SMPs. In ICPADS-12, 2006.

[14] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In ASPLOS-10, 2002.

[15] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In MICRO-36, 2003.

[16] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In ISCA-30, 2003.

[17] T. Sherwood, E. Perelman, B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In PACT-10, 2001.

[18] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In HPCA-11, 2005.

[19] C. CaBcaval and D. A. Padua. Estimating cache misses and locality using stack distances. In ICS-17, 2003.

[20] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In PACT-13, 2004.

[21] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In ISCA-35, 2008