Addressing Fairness in SMT Multicores with a Progress-Aware Scheduler

Josué Feliu, Julio Sahuquillo, Salvador Petit, and José Duato
Department of Computer Engineering (DISCA)
Universitat Politècnica de València
València, Spain
jofepre@gap.upv.es, {jsahuqui,spetit,jduato}@disca.upv.es

***Abstract*—Current SMT (simultaneous multithreading) processors co-schedule jobs on the same core, thus sharing core resources like L1 caches. In SMT multicores, threads also compete among themselves for uncore resources like the LLC (last level cache) and DRAM modules. Per process performance degradation over isolated execution mainly depends on process resource requirements and the resource contention induced by co-runners. Consequently, the running processes progress at different pace. If schedulers are not progress aware, the unpredictable execution time caused by unfairness can introduce undesirable behaviors on the system such as difficulties to keep priority-based scheduling.**

This work proposes a job scheduler for SMT multicores that provides fairness to the execution of multiprogrammed workloads. To this end, the scheduler estimates per-process standalone performance by periodically creating low-contention co-schedules. These estimates are used to compute the per process progress. Then, those processes with less progress are prioritized to enhance fairness.

Experimental results on a Intel Xeon with six dual-threaded SMT cores show that the proposed scheduler reduces unfairness, on average, by 3× over Linux OS. Moreover, thanks to the tread to core allocation policy, the scheduler slightly improves throughput and turnaround time.

***Keywords*-scheduling; fairness; SMT; multicore; performance estimation;**

I. Introduction

Simultaneous multithreading (SMT) [1] allows the processor to exploit both instruction-level and thread-level parallelism. This fact has yield some recent chip multiprocessors (CMPs) like Intel Core and IBM POWER 8 to implement this architectural paradigm. Two kind of shared resources can be distinguished in these systems: intra-core and inter-core resources, which are the shared resources inside the core or in the uncore part of the system, respectively. Shared intra-core and inter-core resources depend on the processor implementation. The instruction queue, the L1 cache and the issue width are typical examples of shared intra-core resources, while the last level cache (LLC) and the main memory are resources commonly shared among cores.

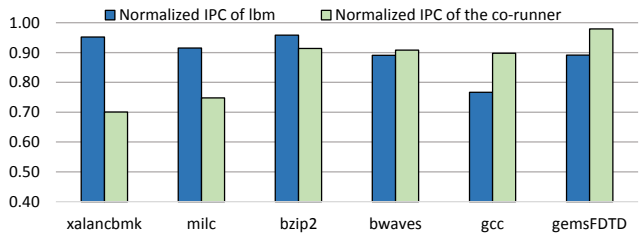
Processes compete among themselves at run time for the shared resources and sharing policies are implemented to regulate their usage. Policies should provide performance fairness to concurrently running applications. Designing fair sharing policies is challenging due to two main issues: i) processes present different requirements for the multiple shared resources, and ii) the shared use of a resource affects differently the performance of the distinct processes.

A system is considered to be fair when all the running processes present the same progress with respect to isolated execution. Unfairness causes important undesirable behaviors on the system [2], [3], [4]: i) it complicates priority-based scheduling since jobs with lower priorities can achieve more progress than those with higher priorities, ii) it makes difficult to guarantee worst-case execution times (WCET), which is particularly important on embedded systems, iii) it reduces performance predictability, which complicates the analysis and optimization of both hardware and software implementations, iv) it can lead to wrong billings in commercial *grid computing* services, where users are charged for CPU hours, and v) it enables denial of service attacks.

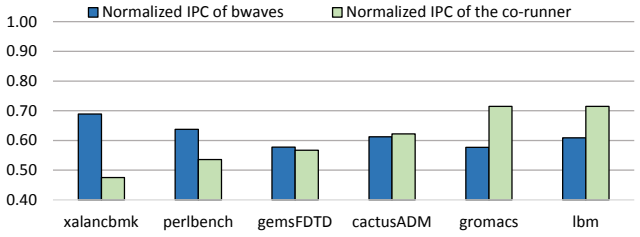
Figure 1 illustrates how inter- (Figure 1a) and intra-core – in addition to inter– (Figure 1b) interferences due to resource sharing can lead to significant unfairness. Figure 1a depicts the normalized instructions per cycle (IPC) of *lbm* over its isolated execution, and the same value for a co-runner (six different co-runners are shown) allocated on another core. Figure 1b presents the results for *bwaves* and another co-runner allocated on the same SMT core. Important progress differences among each possible pair of co-runners, over 30% in both figures, can be observed, although when sharing the same core, differences (in percentage) are much higher. This means that fairness oriented scheduling, considering both intra- and inter-core interferences, is needed.

The key challenge to devise fairness oriented schedulers lies on estimating the progress of each process at run time. Progress can be seen as the number of instructions a process commits running concurrently with respect to the number of instructions it would have committed in isolation during the same period of time. Whereas measuring the instructions completed by the processes in a schedule can be straightforwardly done using performance counters, the difficulty rises in estimating the number of instructions the process would have committed in isolation. A broad range of research work has addressed progress estimation, however, most of them [5], [6], [7] require specific hardware not available in current processors.

In this paper we present the Progress-Aware (PA) sched-



(a) Running with *lbm* on different cores.



(b) Running with *bwaves* on the same SMT core.

Figure 1: Normalized IPC of the benchmarks with respect to isolated execution.

uler for SMT CMPs. The PA scheduler dynamically estimates the isolated performance of a target process at run-time by co-scheduling it on low-contention scenarios. These scenarios remove intra-core interferences for the target process by allocating it on an entire core. At the same time, inter-core interferences are minimized by properly selecting *light-sharing* co-runners. Isolated performance estimates are assumed valid for a given number of quanta. While all the processes have a valid performance estimate, the scheduler addresses fairness, prioritizing those processes with lower accumulated progress. The fact that the system includes SMT cores is leveraged by the scheduler for performance, by properly selecting the pairs of processes to be allocated on each core.

Experimental results on a Intel Xeon E5645 with six dual-threaded SMT cores show that the proposed scheduler reduces unfairness, on average, by 3× over Linux scheduler. Moreover, thanks to the SMT thread allocation policy, turnaround time and throughput are also enhanced up to 6% in some mixes.

The rest of this paper is organized as follows. Section II describes the experimental platform. Section III discusses how the progress made by the processes can be estimated. Section IV presents the progress-aware scheduler. Section V describes the evaluation methodology and Section VI analyzes the experimental results. Section VII goes over the related work. Finally, Section VIII presents some concluding remarks.

II. Experimental platform

All the experimental evaluation has been performed on a Intel Xeon E5645 processor, which consists of six dual-thread SMT cores. Each core includes two levels of private cache, a 32KB L1 and a 256KB L2. A third-level 12 MB cache is shared by all the cores. The system is equipped with 12 GB of DDR3 RAM and runs at 2.4 GHz.

The installed OS is a Fedora Core 10 distribution with Linux kernel 3.11.4. The library *libpfm* 4.3.0 is used to handle hardware performance counters [8]. The scheduler collects at runtime, for each running thread, the processor cycles, the committed instructions, and the number of L1, LLC, and main memory requests. The dynamic information gathered is used by the scheduler to take the corresponding scheduling decisions.

The SPEC CPU2006 benchmark suite with reference inputs has been used in the experiments. For evaluation purposes (see Section V), the target number of instructions for each benchmark is set to the number of instructions executed by the benchmark during 100 seconds in standalone execution.

III. Estimating progress

Accurately estimating how a process progresses at runtime with respect to its isolated execution is the key point to provide fairness in the devised job scheduler. Progress estimation is updated at the end of each quantum for the running processes. For this purpose, we use Equation 1 that accumulates, for the elapsed quanta, the ratio between the measured IPC that a process achieves when running concurrently with other processes ($IPC^{i}_{co-runners}$) and the estimated IPC that such a process would have achieved in isolation (IPC^{i}_{alone}) during the same quantum. The former is directly measured from the `number of instructions` and `number of cycles` gathered with performance counters. The difficulty lies on estimating isolated performance.

$$Progress = \sum_{i=0}^{Q} \frac{IPC^{i}_{co-runners}}{IPC^{i}_{alone}} \quad (1)$$

To estimate standalone IPC of a process, the proposed scheduler arranges a low-contention co-schedule, aimed at minimizing performance interferences among the scheduled processes. The IPC of a target process is measured during the execution of the created low-contention co-schedule and used as estimate of its standalone performance for the n following quanta in which the process is scheduled. During these quanta, the scheduler pursues to provide fairness by prioritizing processes with lower accumulated progress.

Two main reasons can cause deviations in the IPC estimates: i) the standalone IPC is assumed valid for a too long period (number of quanta), and ii) thread interferences are higher than expected. Below these two deviation sources are analyzed.

A. Period length between IPC estimates

Defining the period length between IPC estimates presents a tradeoff between estimation accuracy and fairness. The

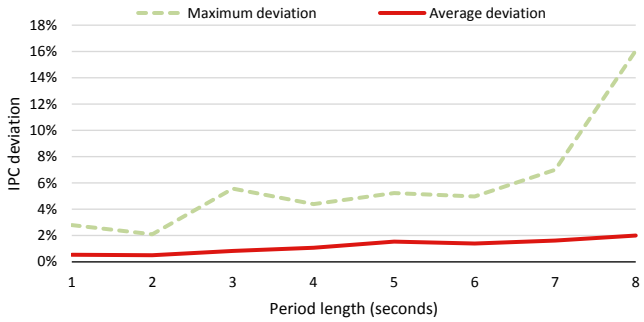


Figure 2: IPC deviation when increasing the period length between measures.

longer the interval, the higher the number of quanta where a given IPC estimate is assumed valid, hence inaccuracy potentially rises. Conversely, the shorter the interval, the higher the number of quanta devoted to IPC estimation; thus, the fewer the quanta used for fairness.

This section analyzes the accuracy of IPC estimates varying the period length. The study compares, for each benchmark, the average deviation of the IPC estimates with respect to the real IPC of each quantum. Average and maximum deviations across all the benchmarks are studied.

Figure 2 presents the results ranging the period length between IPC estimates from 1s to 8s. Solid and dashed lines show the average and maximum deviations across all the benchmarks, respectively. Average values are relatively low (below 2%) for periods shorter than 8s. Maximum deviations, however, grow faster as the period between IPC estimates is enlarged. Nonetheless, results show that reasonable accuracy can be achieved by estimating the standalone IPC of the benchmarks at relatively long periods of time.

To provide further insights in this claim, Figure 3 compares the dynamic IPC evolution of a subset of benchmarks measured at 200ms and 6s periods. When the process presents uniform IPC, like *hmmer*, practically no difference is observed between both sampling periods (in spite that the 6s period is 30× longer than the shorter one), while slight differences can be observed with processes with different phases of execution like *xalancbmk* or *cactusADM*.

These values where obtained running processes alone in the system, however, processes experience a slower (or much slower) progress running with a co-runner on the same SMT core. Therefore, longer periods might be considered in the devised scheduler (see Section IV-C).

B. Process interferences in co-schedules

This section studies performance interferences among processes in the shared resources. The analysis first considers only pairs of benchmarks; then, the co-schedule is extended with more benchmarks to analyze their impact on individual performance. If performance interferences are reasonable, then the standalone IPC may be estimated in co-schedules with multiple co-runners.

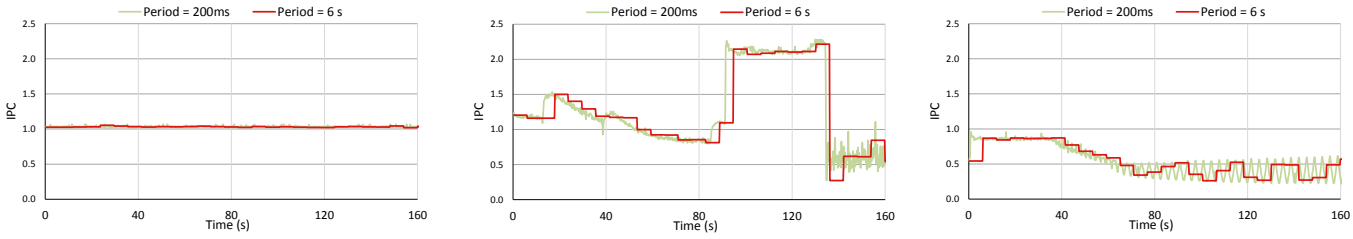
As mentioned above, two levels of interferences are distinguishable in a SMT multicore: intra- and inter-core. Intra-core interferences are caused due to sharing critical core resources for performance like the L1 cache or the dispatch width. In contrast, inter-core interferences are caused by any other running process, which share the last level cache (L3 in our system) and the main memory.

Intra-core interferences impact more strongly on performance since a wider set of resources including L1-caches, instruction queues, and functional units are shared among concurrent threads. Two process that perform scarce use of inter-core resources can run concurrently without noticeable performance degradation. However, intra-core interferences cause any two processes running simultaneously on the same SMT core to significantly reduce their performance. Therefore, to estimate the standalone IPC of a process it should be scheduled alone in a core, avoiding intra-core interferences. From now on, this section focuses on performance interferences among processes running on different cores.

1) Interferences between pairs of benchmarks:

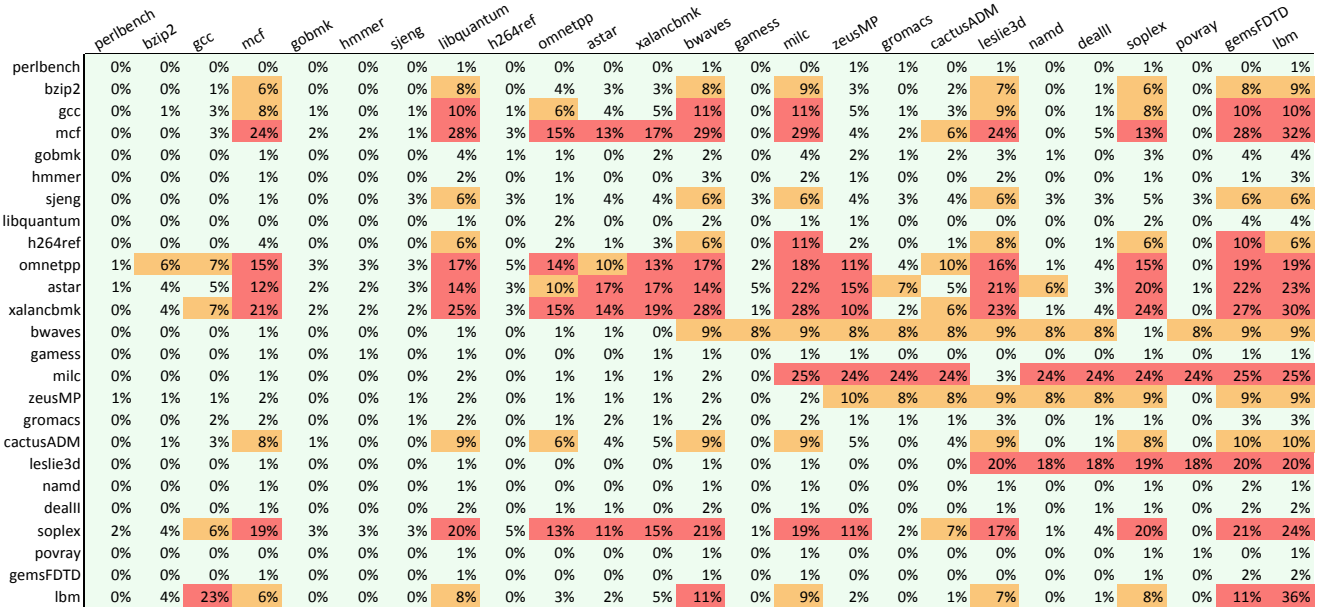
First, the analysis focuses on inter-core interferences between pairs of benchmarks. For this experiment, all the possible couples of benchmarks were ran, each benchmark on a distinct core, and their individual performance compared to that achieved in isolated execution.

Figure 4 presents the results. Each row presents the performance degradation of a benchmark caused by any possible co-runner. For instance, *bzip2* suffers a performance drop by 6% when running with *mcf*, while the performance of *mcf* is not reduced when it is executed with *bzip2*. Similarly, each column presents the performance degradation



(a) hmmer (b) xalancbmk (c) cactusADM

Figure 3: Comparison between IPC measured each 200 ms and each 6 seconds.



	perlbench	bzip2	gcc	mcf	gobmk	hmmer	sjeng	libquantum	h264ref	omnetpp	astar	xalancbmk	bwaves	gamess	milc	zeusMP	gromacs	cactusADM	leslie3d	namd	dealII	soplex	povray	gemsFDTD	lbm
perlbench	0%	0%	0%	0%	0%	0%	0%	1%	0%	0%	0%	0%	1%	0%	0%	1%	1%	0%	1%	0%	0%	1%	0%	0%	1%
bzip2	0%	0%	1%	6%	0%	0%	0%	8%	0%	4%	3%	3%	8%	0%	9%	3%	0%	2%	7%	0%	1%	6%	0%	8%	9%
gcc	0%	1%	3%	8%	1%	0%	1%	10%	1%	6%	4%	5%	11%	0%	11%	5%	1%	3%	9%	0%	1%	8%	0%	10%	10%
mcf	0%	0%	3%	24%	2%	2%	1%	28%	3%	15%	13%	17%	29%	0%	29%	4%	2%	6%	24%	0%	5%	13%	0%	28%	32%
gobmk	0%	0%	0%	1%	0%	0%	0%	4%	1%	1%	0%	2%	2%	0%	4%	2%	1%	2%	3%	1%	0%	3%	0%	4%	4%
hmmer	0%	0%	0%	1%	0%	0%	0%	2%	0%	1%	0%	0%	3%	0%	2%	1%	0%	0%	2%	0%	0%	1%	0%	1%	3%
sjeng	0%	0%	0%	1%	0%	0%	3%	6%	3%	1%	4%	4%	6%	3%	6%	4%	3%	4%	6%	3%	3%	5%	3%	6%	6%
libquantum	0%	0%	0%	0%	0%	0%	0%	1%	0%	2%	0%	0%	2%	0%	1%	1%	0%	0%	0%	0%	0%	2%	0%	4%	4%
h264ref	0%	0%	0%	4%	0%	0%	0%	6%	0%	2%	1%	3%	6%	0%	11%	2%	0%	1%	8%	0%	1%	6%	0%	10%	6%
omnetpp	1%	6%	7%	15%	3%	3%	3%	17%	5%	14%	10%	13%	17%	2%	18%	11%	4%	10%	16%	1%	4%	15%	0%	19%	19%
astar	1%	4%	5%	12%	2%	2%	3%	14%	3%	10%	17%	17%	14%	5%	22%	15%	7%	5%	21%	6%	3%	20%	1%	22%	23%
xalancbmk	0%	4%	7%	21%	2%	2%	2%	25%	3%	15%	14%	19%	28%	1%	28%	10%	2%	6%	23%	1%	4%	24%	0%	27%	30%
bwaves	0%	0%	0%	1%	0%	0%	0%	1%	0%	1%	1%	0%	9%	8%	9%	8%	8%	8%	9%	8%	8%	1%	8%	9%	9%
gamess	0%	0%	0%	1%	0%	1%	0%	1%	0%	0%	0%	1%	1%	0%	1%	1%	0%	0%	0%	0%	0%	1%	0%	1%	1%
milc	0%	0%	0%	1%	0%	0%	0%	2%	0%	1%	1%	1%	2%	0%	25%	24%	24%	24%	3%	24%	24%	24%	24%	25%	25%
zeusMP	1%	1%	1%	2%	0%	0%	1%	2%	0%	1%	1%	1%	2%	0%	2%	10%	8%	8%	9%	8%	8%	9%	0%	9%	9%
gromacs	0%	0%	2%	2%	0%	0%	1%	2%	0%	1%	2%	1%	2%	0%	2%	1%	1%	1%	3%	0%	1%	1%	0%	3%	3%
cactusADM	0%	1%	3%	8%	1%	0%	0%	9%	0%	6%	4%	5%	9%	0%	9%	5%	0%	4%	9%	0%	1%	8%	0%	10%	10%
leslie3d	0%	0%	0%	1%	0%	0%	0%	1%	0%	0%	0%	0%	1%	0%	1%	0%	0%	0%	20%	18%	18%	19%	18%	20%	20%
namd	0%	0%	0%	1%	0%	0%	0%	1%	0%	0%	0%	0%	1%	0%	1%	0%	0%	0%	1%	0%	0%	1%	0%	2%	1%
dealII	0%	0%	0%	1%	0%	0%	0%	2%	0%	1%	1%	0%	2%	0%	1%	0%	0%	0%	1%	0%	1%	1%	0%	2%	2%
soplex	2%	4%	6%	19%	3%	3%	3%	20%	5%	13%	11%	15%	21%	1%	19%	11%	2%	7%	17%	1%	4%	20%	0%	21%	24%
povray	0%	0%	0%	0%	0%	0%	0%	1%	0%	0%	0%	0%	1%	0%	1%	0%	0%	0%	0%	0%	0%	1%	1%	0%	1%
gemsFDTD	0%	0%	0%	1%	0%	0%	0%	1%	0%	0%	0%	0%	1%	0%	1%	0%	0%	0%	0%	0%	0%	1%	0%	2%	2%
lbm	0%	4%	23%	6%	0%	0%	0%	8%	0%	3%	2%	5%	11%	0%	9%	2%	0%	1%	7%	0%	1%	8%	0%	11%	36%

Figure 4: Performance degradation due to inter-core interferences running pairs of benchmarks. Each row shows the degradation of a benchmark running with each co-runner on differents cores.

a benchmark induces to each co-runner. For instance, among all the possible co-runners, *libquantum* causes the highest performance drop (by 28%) to *mcf*.

The performance degradation level is highlighted in the table with different colors (or levels of gray). A cell (X,Y) colored in green (light gray), orange (medium gray), or red (dark gray), means that process Y affects the performance of process X less than 5%, between 5% and 10%, or more than 10%, respectively.

Depending on how benchmarks affect the performance of their co-runners, they can be classified in two main categories: *heavy-sharing* and *light-sharing*. The former category includes benchmarks that strongly (e.g. above 10%) affect the performance of a significant subset of the possible co-runners. Examples of benchmarks belonging to this category are *mcf*, *libquantum* and *omnetpp*. The *light-sharing* category includes those benchmarks that scarcely affect the performance of the co-runners since they make a scarce use of the shared resources. This category includes benchmarks in columns that mostly include cells colored in green.

Note that for any target benchmark, a wide set of co-runners impacting its performance less than 5% can be found. For example, *perlbench* can be coupled to estimate its standalone IPC with any other benchmark since the maximum performance degradation it suffers is by 1%. Following the same rule, *astar* can be paired with *perlbench*, *bzip2*, *gcc*, *gobmk*, etc.

A scheduler could use the above off-line analysis to predict performance interferences. However, this way is impractical in a real system. In contrast, our approach consists on classifying benchmarks as heavy-sharing or light-sharing at run-time. After a wide set of experiments analyzing distinct performance counters, we found that the bandwidth consumption of the uncore shared resources, that is, the LLC and the main memory (MM), are appropriate metrics to match this classification.

At a first glance, it might be expected that processes with either high LLC bandwidth or high MM bandwidth consumption fall on the heavy-sharing category since they are likely to interfere the co-runners performances. Conversely, processes that perform a scarce use of these resources are unlikely to interfere with co-runners, so they could be classified as light-sharing.

Figure 5 depicts the average MM and LLC bandwidth consumption of the benchmarks in standalone execution. As observed, all the benchmarks whose LLC bandwidth utilization is above 19 trans/μs or whose MM bandwidth utilization is above 3.5 trans/μs belong to the heavy-sharing category. Otherwise, they fall in the light-sharing category. Notice that these thresholds are estimated in standalone execution and cache interference when the processes run concurrently can cause cache misses to grow. Thus, it is likely that processes with bandwidth utilizations close to the thresholds (e.g., *gcc* or *cactusADM*) will exceed them when running with other processes.

2) Cumulative interferences in low-contention co-schedules:

The previous section analyzed interferences in low-contention co-schedules composed only of a pair of co-runners. However, to avoid significant throughput reduction when IPC estimates are required, the number of light-sharing benchmarks executing in a low-contention co-schedule should be as high as possible.

To analyze cumulative interferences in low-contention

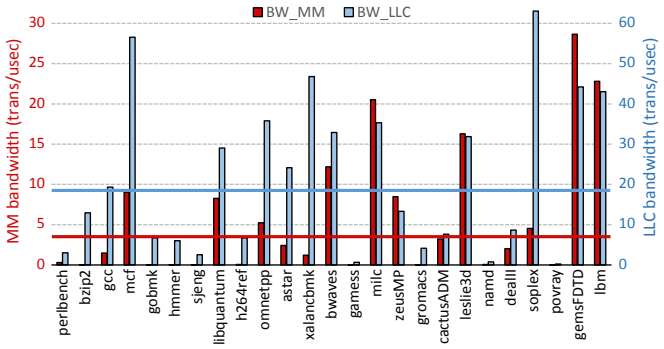


Figure 5: Average main memory and LLC bandwidth.

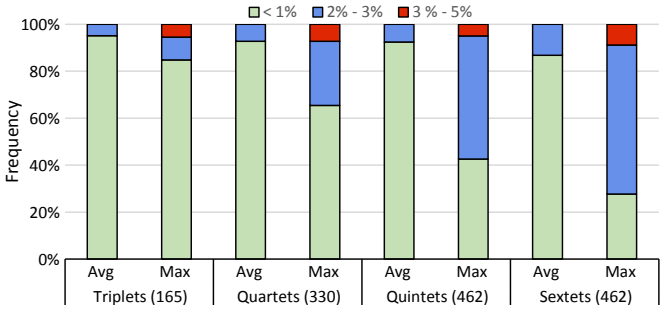


Figure 6: Histogram of the performance degradation on light-sharing co-schedules. In brackets, the total number of evaluated co-schedules.

co-schedules consisting of more than two co-runners, the performance of all possible groups of three, four, five, and six light-sharing benchmarks has been explored. Figure 6 depicts the results. Looking the average performance slowdown (Avg bar), it can be observed that performance interferences are acceptable even in large groups. For instance, more than 85% of all the possible (i.e. 462) 6-process (sextets) co-schedules present an average slowdown below 1%. The Max bar refers to the slowdown of the benchmark suffering the highest slowdown in the co-schedule. As expected, it grows as the number of processes of the co-schedules rises. However, only by 10% of the benchmarks in the 6-process co-schedules present a maximum performance degradation falling in between 3% and 5%.

IV. Progress-Aware Scheduling

The proposed PA scheduler is designed to allow all the processes to achieve the same progress over the mix execution. The impact of interferences on individual process performance widely differs across the studied processes. This means that processes require distinct execution times to achieve the same progress. In other words, processes with higher performance degradation induced by co-runners require more quanta of execution than processes with lower degradation to achieve equal progress.

As mentioned above, the scheduler uses some quanta to estimate isolated performance of a target process. These quanta can affect both fairness and performance due to scheduling constraints when creating low-contention co-schedules. For instance, in low-contention co-schedules light-sharing processes are prioritized over heavy-sharing processes regardless their accumulated progress. Remark that processes are dynamically broken down at run time as heavy- or light-sharing according to their bandwidth consumption in the uncore resources (LLC and main memory) during the last quantum, which are updated using performance counters.

The scheduler can work on two different modes: IPC estimation-oriented and fairness-oriented. The former applies when any process needs to estimate its isolated IPC and a low-contention co-schedule is required. The latter guides the scheduling to improve fairness and applies when all the processes have a valid IPC estimate. Algorithm 1 presents the pseudocode of the progress-aware scheduler, which differentiates between both scheduling *modes*: IPC estimation-oriented mode (lines 1 to 9) and fairness-oriented mode (lines 10 to 16).

IPC estimates for each process are kept valid for a certain number n (see Section IV-C) of quanta. A saturating counter is assigned to each P process to account the elapsed quanta, and is updated each quantum the process is scheduled. When any counter saturates, the scheduler moves to the IPC estimation-oriented mode and the counter is reset. Otherwise, the fairness-oriented mode determines the co-schedules for the following quanta. Below these two modes are described, and then the implementation parameters are discussed.

A. IPC estimation-oriented mode

The IPC estimation-oriented mode (lines 2 to 10 of the algorithm) is triggered when a valid IPC estimate is required

Algorithm 1 Progress-Aware scheduler

1: **if** the IPC estimation of any process P *has expired* **then**

IPC-ESTIMATION MODE

2: Allocate P to an entire core.
3: **if** P is a light-sharing process **then**
4: **while** IPC estimation of any light-sharing process P_{LS} is *close to expire* **and** there are free cores **do**
5: Allocate P_{LS} to an entire core.
6: **end while**
7: **end if**
8: Select as many light-sharing processes as available hardware threads, prioritizing those with lower progress.
9: Allocate pairs of processes to free cores using [9]
10: **else**

FAIRNESS MODE

11: **while** a process P_{LP} is progressing *below the average* **do**
12: Allocate P_{LP} to an entire core.
13: **end while**
14: Select as many processes as available hardware threads prioritizing those with lowest progress.
15: Allocate pairs of processes to free cores using [9]
16: **end if**

(line 1) for any process *P*. A low-contention scenario is scheduled to avoid intra-core and minimize inter-core interferences. The former interferences are removed by allocating *P* to an entire core (line 2). Inter-core interferences are minimized by only including light-sharing processes in the co-schedule.

If *P* itself is a light-sharing process (line 3), and there are other light-sharing processes whose IPC estimates are *close to expire* (see Section IV-C), then each of them is allocated to an individual core (line 5). This way allows multiple IPC estimates to be obtained during the same quantum.

After that, the remaining cores are filled with light-sharing processes. In particular, as many light-sharing processes as available SMT hardware threads are co-scheduled (line 8). For the sake of fairness, the scheduler prioritizes the light-sharing processes that have experienced less accumulated progress. These processes are smartly allocated to cores in pairs to reduce SMT intra-core interferences according to [9]. This thread-to-core allocation strategy considers the L1-cache bandwidth consumption of the processes. Finally note that if there are not enough light-sharing processes in the workload, the exceeding hardware threads are left idle.

B. Fairness-oriented mode

As a rule of thumb, to improve fairness, the scheduler selects those processes with lowest progress to run the following quantum (line 14). On the experimental platform, the twelve processes with lowest progress are selected. Nonetheless, to maximize fairness, this execution mode checks, in a prior step, if the progress of any process is falling behind the others. To this end, the scheduler computes the average progress of all the processes. Then, it is tested if the progress of any process is 5% below the average (line 11). If there are some processes in this situation, the scheduler allocates each of them to an individual core (line 12). This way speedups their progress since individual execution is faster than when sharing the core. After that, the scheduler selects the remaining processes and allocates pairs of them on the remaining cores considering their L1-cache bandwidth consumption.

Finally, note that even when working in this mode, the scheduler can take profit of unprompted scenarios to estimate isolated IPCs. For instance, if a co-schedule only includes light-sharing processes, isolated performance can be estimated for those processes individually allocated to a core, regardless the deadline of their IPC estimate.

C. Implementation considerations

The proposed algorithm relies on several parameters, that must be tuned to provide its best results. Depending on the values of these parameters the schedulers can: i) maximize fairness with no performance considerations, ii) prioritize fairness over (but without compromising) performance. Different values for these parameters have been evaluated. This section presents and discusses the values used to evaluated the proposal, analyzing their advantages and disadvantages.

The maximum period between two standalone IPC estimates has been empirically set to 8 seconds, that is $n = 40$ 200ms quanta. Experimental results showed that shorter periods could enhance fairness, but strongly affecting the performance. Conversely, longer intervals negatively affect fairness without providing significant performance benefits. In the algorithm implementation, we also consider that an isolated IPC estimation is *close to expire* when the number of quanta a process has been scheduled since its last estimation is half ($n = 20$) the maximum number of quanta.

Main memory and LLC bandwidth thresholds to discern between light- and heavy-sharing processes are set to 3.5 and 19 tran/μs, respectively, since these values offer a good trade-off between fairness and performance. Higher thresholds include more benchmarks classified as light-sharing even if they are not (i.e., they produce considerable contention). As a consequence, more contention than expected can be generated so system fairness can be compromised. On the contrary, lower thresholds classify more processes as heavy-sharing. However, in this case, performance may be affected since a higher number of heavy-sharing processes limits the scheduler flexibility.

The last parameter used in the algorithm determines when a given process is unfairly *progressing below the others*. As explained before, when this situation occurs, the scheduler allocates the process to an entire core to accelerate its progress, avoiding inter-core interferences. We consider that a process is progressing too slow when its progress differs above 5% from the average progress of the processes of the mix. Using a higher threshold would enlarge the accepted unfairness before taking scheduling decisions to reduce it. Conversely, a lower threshold would trigger the progress *correction* too frequently, so affecting the system performance.

V. Evaluation methodology

A. Scheduler implementation

To evaluate the effectiveness of the progress-aware scheduler, we compare its fairness and performance to those achieved by the Linux scheduler. The proposed algorithm is implemented in a user-level scheduler. Performance counters are used to update IPC and MM-, LLC- and L1-bandwidth consumption of the different processes at runtime. The Linux system calls and the thread-to-core affinity attribute of the processes are used to co-schedule the selected processes. The former determines which processes run at a given quantum, and the latter, their allocation to cores.

Linux schedules are also evaluated with a user-level scheduler to monitor the number of instructions each process executes (see Section V-B). To *mimic* the Linux behavior from this user-level scheduler, all the processes are allowed

to run each quantum on any core, so the Linux kernel scheduler must determine the actual co-schedule (both process selection and allocation) [10], [11].

The overhead arising from the algorithm implementation is negligible considering the 200ms quantum length at which scheduling is performed. Overall overhead, including process selection, process allocation and progress accounting, as well as, processes and performance counters management, is by 0.1 ms. Note that it is below 1 ‰ of the quantum length.

B. Mix design

A set of fourteen mixes composed of twenty-four SPEC CPU2006 benchmarks has been designed to evaluate the proposed algorithm. Each mix consists of a variety of light- and heavy-sharing benchmarks. Mixes have been sorted according to their percentage of heavy-sharing benchmarks, which ranges from 25% to 60%.

Since SPEC CPU2006 benchmarks present widely different execution times. To equalize the differences, we run each benchmark for a target number of instructions, which corresponds to the amount of instructions it executes during 100s (i.e. 500 quanta) in isolation. Benchmarks with shorter or longer execution time are relaunched or killed, respectively, to run exactly their target number of instructions. In this way, we avoid benchmarks to present different weights during the mix execution.

C. Metrics

Fairness can be achieved at the cost of performance. Therefore, it cannot be evaluated in isolation, but performance metrics should also be considered.

Running multiprogrammed workloads, fairness metrics estimate if performance benefits or losses are similar across all the processes, and do not concentrate only on a few of them. Recent work has used the unfairness metric [12], [13], [14] for this purpose. This metric is defined as the maximum slowdown divided by the lowest slowdown across all the processes (N) of the workload as shown in Equation 2. Notice that an unfairness equal to 1 means that the system is completely fair.

$$Unfairness = \frac{Max\ Slowdown_i}{Min\ Slowdown_j}\ \forall\{i,j\} \in \{1,N\} \quad (2)$$

The slowdown of each process is calculated as the ratio between its elapsed execution time in the co-schedule and its standalone execution time as shown in Equation 3. The elapsed execution time of a process i not only accounts for the time the process is actually running ($Ti^{running}_{Workload}$), but also for those quanta the process is not scheduled ($Ti^{waiting}_{Workload}$).

$$Slowdown_i = \frac{Ti^{running}_{Workload} + Ti^{waiting}_{Workload}}{Ti_{alone}} \quad (3)$$

A wide set of metrics has been used on recent research work to evaluate the performance of multiprogrammed workloads running on multicores, but it is still in debate what are the best ones [15]. This work evaluates turnaround time and system throughput (STP). The impact on both turnaround time and STP is analyzed to explore how performance is affected by the PA scheduler. Turnaround time is a user-oriented performance metric and measures the elapsed time since the workload is launched to execution until the last process of the workload completes its execution. STP quantifies the system-level throughput and it is computed as the sum of the weighted IPC of the processes of the workload [15]. The weighted IPC of a process is the ratio between its IPC when it is co-scheduled and its standalone IPC.

VI. Experimental evaluation

A. Fairness

This section evaluates the fairness of the PA scheduler and the Linux scheduler. Figure 7 depicts the unfairness, in percentage, presented by both Linux and the proposed scheduler across the evaluated mixes. For each mix, the figure presents the average values of 20 executions with both schedulers and a 95% confidence interval.

The PA scheduler is fairer than Linux across all the mixes. Unfairness ranges in a relatively narrow interval, from 8% to 18%, with an average by 12%. In contrast, Linux unfairness ranges in a wide interval, from 19% to 44%, with an average by 32%. This means that under Linux, the slowest process progresses on average by 32% slower than the fastest one. These values seem high and inappropriate in many real systems (e.g. they complicate priority-based scheduling or the estimation of WCET in real-time systems). Compared to Linux, PA reduces unfairness, on average, by a 3× factor under the stduied mixes. In addition, the presented 95% confidence intervals show the steadiness of the unfairness values through multiple executions of each mix.

Taking into account that mixes have been sorted by the number of heavy-sharing processes they include, results suggest that, in general, Linux achieves lower levels of unfairness when contention is lower. On the contrary, the unfairness provided by the PA scheduler tends to be more

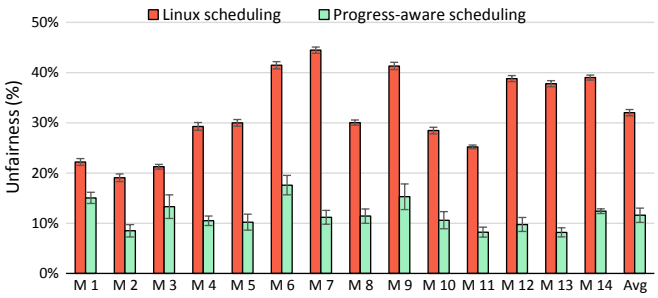


Figure 7: Unfairness (lower is better).

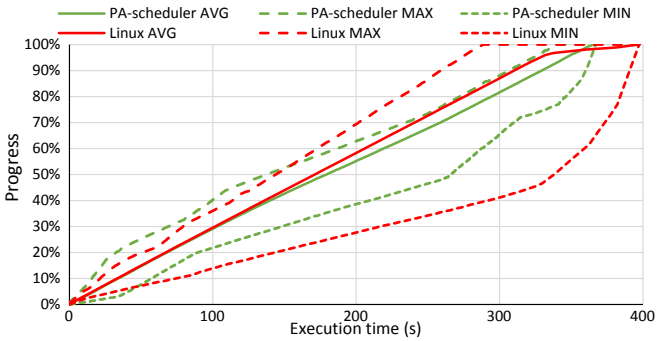


Figure 8: Dynamic progress of processes in mix M7.

uniform, regardless the number of heavy-sharing processes considered in the studied mixes. Therefore, we can conclude that the higher the contention, the higher the fairness benefits the PA scheduler provides over Linux.

Figure 8 focuses on mix M7 to illustrate how unfairness evolves over the mix execution. Average, maximum, and minimum progress achieved by the processes for the Linux and the PA schedulers are plotted. Remark that in this figure, real progress is plotted since it is calculated and not estimated (only to show the progress, not to guide scheduling) for each process as the ratio between committed instructions and the target number of instructions to be committed.

Results depict how Linux unfairness grows with time. For instance, when the first process of the mix finishes at time 280s under Linux, the process with lowest progress has only completed by 40% of its execution. In contrast, the PA scheduler handles progress more uniformly across all the processes. At time 340s, when the first process finishes, the process with lowest progress has committed about 75% of its instructions. Moreover, there is a bigger gap between the maximum and minimum progress with the Linux scheduler for most of the execution time.

B. Performance

Figure 8 also shows that the PA scheduler slightly improves turnaround time over Linux, since the complete workload finishes its execution sooner than under Linux. This section analyzes this performance improvement.

Figure 9a presents the speedup of the turnaround time achieved by the PA scheduler over Linux across the studied mixes. Results show that despite the main focus is on fairness, the PA scheduler improves turnaround time across all the mixes. This improvement is above 3.5% in five mixes, and by 7% in mix M6. The reason is that the PA scheduler allows all the processes to progress at similar rate, and consequently, the workload execution finishes sooner. In contrast, if the scheduler is not progress aware, the process with lowest progress will make the mix execution time longer.

Figure 9b presents the speedup of the STP achieved by the PA scheduler with respect to Linux. The proposed scheduler improves Linux STP ranging from 2% to 6%, with a geometric mean of 3.4%. These speedups demonstrate that the system throughput is not adversely but positively affected by the PA scheduler. The key reason is that processes are selected for fairness but allocated to cores for performance. Effectively, the scheduler addresses performance by choosing the proper pair of threads to be allocated in the same SMT core reducing intra-core interferences [9].

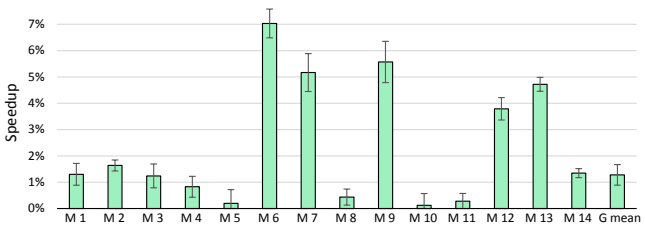
C. Accuracy of the isolated IPC estimations

Accurate IPC estimates are required to improve fairness. If these estimates are inaccurate, the computed progress will differ from the actual progress, which will yield unfairness to grow.

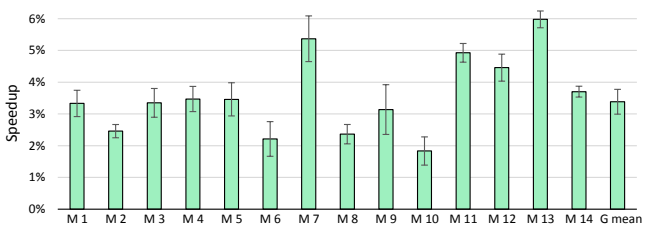
Figure 10 presents the average, maximum, and minimum IPC accuracy across the twenty four processes of each mix. Results show that average IPC accuracy ranges from 95% to 98%, which confirms that the proposed mechanism is able to correctly estimate isolated performance of the processes in co-schedules. Notice that by 100% accuracy is always achieved by at least one process of each mix. This is due to the fact that some processes present a uniform IPC across its execution time, which helps accurate estimates. Regarding maximum deviation from the real IPC, accuracy ranges from 82% to 93%.

VII. Related work

Contention in shared resources has been addressed in scheduling algorithms, but an important piece of this work in the past [11], [10], [16], [9] mainly focus on performance without taking fairness into account.



(a) Speedup of the turnaround time.



(b) Speed up of the system throughput (STP).

Figure 9: Performance benefits over Linux.

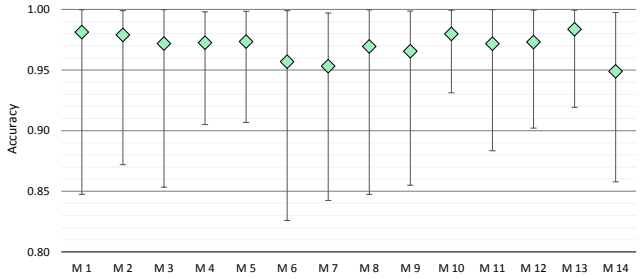


Figure 10: Average, maximum, and minimum accuracy of the isolated IPC estimations.

This paper is mainly related to two research topics: fairness techniques and performance predictability. These topics have been already addressed together in previous works, where performance predictability models have been used to estimate fairness in either a shared resource or the system.

Several works have addressed fairness from a shared resource perspective, trying to provide fair sharing in a given resource. Some of them have focused on uncore memory resources. [2] and [17] concentrate on the memory controller to improve the system fairness. In [2], Mutlu et al. propose a memory access scheduler that balances the DRAM-related slowdown experienced by the co-scheduled processes. A similar approach is followed in [17], where Nesbit et al. use concepts from network fair queuing to design a fair queuing memory system. Finally, Ebrahimi et al. [12] propose achieving fairness via source throttling, a global mechanism that addresses unfairness on the entire shared memory system.

Other works deal with fairness in SMT fetch policies. Luo et al. [18] and Eyeraman et al. [19] propose SMT fetch policies that enhance both performance and fairness considering the pipeline status and memory-level parallelism, respectively.

Cache partitioning techniques also try to provide a fair cache access to the processes sharing the same cache structure. Suh et al. [20] estimate the isolated miss-rate of the processes to improve the partitioning, and Kim et al. [21] dynamically partition L2 caches based on metrics that correlate with execution-time fairness.

Unfortunately, fairly sharing a single resource or a set of resources does not provides fairness to the system. Therefore, other authors address aims to provide system fairness, by focusing on process scheduling. Fairness oriented process schedulers have been proposed by Fedorova et al. [22] and Xu et al. [13]. In [22], Fedorova et al. target shared-cache contention using resource performance. Xu et al. [13] mainly target main memory contention and focus on overall system fairness with a scheduler that monitors the progress of the processes at runtime.

Compared to these works, the proposed PA scheduler tackles fairness on SMT multicores, considering both intra- and inter-core interferences to provide a fair execution among the different processes of a workload. In addition, the proposed scheduler also addresses performance when allocating processes to cores, as done in [9], which allows simultaneous performance and fairness enhancements in SMT platforms.

Regarding performance predictability, in [5] and [6], Eyerman et al. propose cycle accounting architectures that allow accurate predictions of the isolated performance of the processes while they run concurrently on out-of-order CMPs and SMT processors, respectively. The proposed models can be used to design a process scheduler targeting system fairness. An orthogonal solution was proposed by Cazorla et al. [3] allowing the OS to run jobs at a certain percentage of their maximum speed, regardless of the system load.

Finally, Subramanian et al. [7] combine performance predictability with fairness-oriented main-memory request scheduling. Authors first present a model that estimates the slowdowns caused by memory interferences by modifying the priority scheme of the memory controller. Then, they use this model as the base of two memory request scheduling schemes that provide quality-of-service and maximize fairness, respectively.

VIII. Conclusions

Fairness-aware scheduling is gaining importance in multicore systems to guarantee correct management of process priorities, quality of service, worst case execution times, energy consumption, etc. Allocating the same execution time and resources to the running processes in a multiprogrammed workload does not provide fairness because of the unpredictable interferences on the shared resources. Several techniques used in current systems, such as heterogeneous cores, dynamic voltage and frequency scaling (DVFS), application-specific hardware accelerators, etc. complicate even more fairness-oriented scheduling.

This work has presented the PA progress-aware scheduler for SMT multicores. The main challenge to achieve fairness lies on dynamically and accurately estimating at run-time the progress of each process over isolated execution. To accomplish this, the proposed scheduler periodically creates low-contention co-schedules where the isolated performance of the processes can be estimated. The scheduler presents two working modes: IPC estimation and fairness-oriented. When all the processes have a valid estimate, the scheduler prioritizes those processes with the lowest relative progress to achieve fairness.

The PA scheduler exploits multithreaded cores by allocating a single process to an entire core or by allocating a pair of processes to that core, mainly depending on the working mode. An important observation is that performance is not necessarily damaged by this fact, since a process can speedup its execution much faster running alone on a core than when sharing the core with a co-runner.

Experimental evaluation results obtained in a Intel Xeon E5645 show that the PA scheduler improves unfairness by a 3× factor with respect to Linux. In addition, thanks to the SMT thread to core allocation policy, turnaround time and throughput are also enhanced up to 6% in some mixes.

ACKNOWLEDGMENTS

This work was supported by the Spanish Ministerio de Economía y Competitividad (MINECO) and Plan E funds, under Grant TIN2012-38341-C04-01, and by the Intel Early Career Faculty Honor Program Award.

REFERENCES

[1] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *SIGARCH Comput. Archit. News*, vol. 23, no. 2, pp. 392–403, May 1995.

[2] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *International Symposium on Microarchitecture (MICRO)*, 2007, pp. 146–160.

[3] F. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero, "Predictable Performance in SMT Processors: Synergy Between the OS and SMTs," *IEEE Transactions on Computers*, vol. 55, no. 7, pp. 785–799, 2006.

[4] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *16th USENIX Security Symposium*. USENIX, August 2007.

[5] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A Performance Counter Architecture for Computing Accurate CPI Components," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 175–184.

[6] S. Eyerman and L. Eeckhout, "Per-thread Cycle Accounting in SMT Processors," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009, pp. 133–144.

[7] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *International Conference on High-Performance Computer Architecture (HPCA)*, 2013, pp. 639–650.

[8] S. Eranian, "What Can Performance Counters Do for Memory Subsystem Analysis?" in *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2008, pp. 26–30.

[9] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "L1-Bandwidth Aware Thread Allocation in Multicore SMT Processors," in *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2013, pp. 123–132.

[10] D. Xu, C. Wu, and P.-C. Yew, "On Mitigating Memory Bandwidth Contention Through Bandwidth-Aware Scheduling," in *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2010, pp. 237–248.

[11] J. Feliu, S. Petit, J. Sahuquillo, and J. Duato, "Cache-Hierarchy Contention Aware Scheduling in CMPs," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, March 2014, pp. 581–590.

[12] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 335–346.

[13] D. Xu, C. Wu, P.-C. Yew, J. Li, and Z. Wang, "Providing Fairness on Shared-memory Multiprocessors via Process Scheduling," in *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2012, pp. 295–306.

[14] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi, "Application-to-Core Mapping Policies to Reduce Memory Interference in Multi-core Systems," in *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2012, pp. 455–456.

[15] S. Eyerman and L. Eeckhout, "Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance," in *Computer Architecture Letters*, vol. 13, no. 2, 2014, pp. 93–96.

[16] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M.-L. Soffa, "The Impact of Memory Subsystem Resource Sharing on Datacenter Applications," in *International Symposium on Computer Architecture (ISCA)*, 2011, pp. 283–294.

[17] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair Queuing Memory Systems," in *International Symposium on Microarchitecture (MICRO)*, 2006, pp. 208–222.

[18] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Throughput and Fairness in SMT Processors," in *International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2001, pp. 164–171.

[19] S. Eyerman and L. Ecckhout, "A Memory-Level Parallelism Aware Fetch Policy for SMT Processors," in *International Conference on High-Performance Computer Architecture (HPCA)*, Feb 2007, pp. 240–249.

[20] G. Suh, S. Devadas, and L. Rudolph, "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," in *International Conference on High-Performance Computer Architecture (HPCA)*, 2002, pp. 117–128.

[21] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," in *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2004, pp. 111–122.

[22] A. Fedorova, M. Seltzer, and M. D. Smith, "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler," in *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2007, pp. 25–38.