Perf&Fair: a Progress-Aware Scheduler to Enhance Performance and Fairness in SMT Multicores

Josué Feliu, Julio Sahuquillo, *Member, IEEE,* Salvador Petit, *Member, IEEE,* and José Duato

Abstract—Nowadays, high performance multicore processors implement multithreading capabilities. The processes running concurrently on these processors are continuously competing for the shared resources, not only among cores, but also within the core. While resource sharing increases the resource utilization, the interference among processes accessing the shared resources can strongly affect the performance of individual processes and its predictability. In this scenario, process scheduling plays a key role to deal with performance and fairness.
In this work we present a process scheduler for SMT multicores that simultaneously addresses both performance and fairness. This is a major design issue since scheduling for only one of the two targets tends to damage the other. To address performance, the scheduler tackles bandwidth contention at the L1 cache and main memory. To deal with fairness, the scheduler estimates the progress experienced by the processes, and gives priority to the processes with lower accumulated progress.
Experimental results on an Intel Xeon E5645 featuring six dual-threaded SMT cores show that the proposed scheduler improves both performance and fairness over two state-of-the-art schedulers and the Linux OS scheduler. Compared to Linux, unfairness is reduced to a half while still improving performance by 5.6%.

Index Terms—scheduling, fairness, SMT, multicore, performance estimation.

1 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 i7 and IBM POWER8 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 within the core or in the uncore part of the system, respectively. Shared intra-core and inter-core resources vary with the processor architecture. 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 them at run time for shared resources and sharing policies are implemented to regulate their usage. These policies should provide performance and fairness to concurrently running applications. However, designing fair sharing policies is challenging due to two main issues. First, processes present different requirements for the multiple shared resources, and second, the shared use of a resource affects differently the individual performance of the distinct processes.

Several notions of fairness can be found in the literature. In this paper, a system is considered fair when all the running processes present the same slowdown with respect to their 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, and iv) it enables denial of service attacks.

Several proposals try to provide fairness from a resource perspective, such as the memory controller [2], [5] or the shared caches [6]. Unfortunately, fairly sharing a resource does not guarantee system fairness. Thus, other proposals are built over the concept of progress [7], as a way of providing system fairness. The key challenge of this kind of 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 running 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.

Despite the current relevance of fairness, it is not commonly acceptable to improve it at expense of overall workload performance. However, targeting fairness and performance at the same time is not an easy task. For example, a prevalent approach to improve performance consists in balancing the memory requests of a multiprogrammed workload along its execution time [8], [9]. In contrast, to improve fairness, the processes with less accumulated progress can be given priority over processes with superior progress. Unfortunately, both strategies can easily conflict. In such a case, preference should be given to one of the targets

• *The authors are with the Department of Computer Engineering (DISCA), Universitat Politècnica de València, Camí de Vera s/n, València 46022, Spain. E-mail: jofepre@gap.upv.es, {spetit,jsahuqui,jduato}@disca.upv.es*

(performance or fairness), penalizing the other.

The main contribution this paper makes is the design and implementation of the *Perf&Fair* scheduler, a scheduling algorithm that addresses both performance and fairness in multicores consisting of SMT cores when the number of processes exceeds the number of hardware threads. From the best of our knowledge this is the first scheduling algorithm on a real system that explicitly addresses performance and fairness simultaneously. Notice that the goal of this work is not to replace the kernel OS scheduler, but allow it to use the proposed policy when the target workload comprises CPU and memory intensive applications such as the SPEC CPU benchmarks.

To deal with fairness, the proposed scheduler estimates the progress that processes experience at runtime. This is done using estimates of their standalone IPC, which are periodically obtained running the processes in low-contention co-schedules. To address performance, the proposal tackles bandwidth contention at the main memory and L1 caches using a bandwidth-aware scheduling algorithm [8], [9].

Experimental results on a Intel Xeon E5645 with SMT cores show that, compared to the Linux OS scheduler, the proposed scheduler reduces unfairness on average to a half. At the same time, the performance in terms of turnaround time is enhanced by 6% (geometric mean).

The rest of this paper is organized as follows. Section 2 describes the experimental platform. Section 3 discusses how the progress made by the processes is estimated. Section 4 summarizes previous work on bandwidth-aware scheduling. Section 5 presents the *Perf&Fair* scheduler. Section 6 describes the evaluation methodology and Section 7 analyzes the experimental results. Section 8 goes over the related work. Finally, Section 9 presents some concluding remarks.

2 Experimental Platform

All the experimental evaluation has been performed on a Intel Xeon E5645 processor with six dual-thread SMT cores. Each core includes two levels of private caches, a 32KB L1 cache and a 256KB L2 cache. A third-level 12 MB cache is shared by all the cores. The system is equipped with 12 GB of DDR3 RAM, runs at 2.4 GHz, and the Intel Turbo Boost mode is disabled to prevent uncontrolled frequency increases when only one thread is running on a core.

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 [10]. The devised scheduler collects at runtime for each running thread: the processor cycles, the committed instructions, and the number of L1, LLC, and main memory requests. This information is used by the scheduler to guide scheduling decisions.

The SPEC CPU2006 benchmark suite with reference inputs has been used in the experiments. For evaluation purposes (see Section 6), 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.

3 Estimating Progress

Accurately estimating how a process progresses at runtime with respect to its isolated execution is a key challenge to provide fairness in the devised job scheduler. For estimating progress, we use Equation 1 that accumulates, for the elapsed execution time (in steps of quanta), the ratio between the measured IPC that a process achieves running alongside 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 amount of time. The former is directly measured using the number of committed instructions and number of cycles gathered with the available 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 the performance interference among the scheduled processes. The IPC of a target process is measured during the execution of the low-contention co-schedule and used as estimate of its standalone performance for the n following quanta in which the process is scheduled. 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. The *Perf&Fair* scheduler proposed in this work uses a 8 seconds period length between estimates, and bandwidth thresholds of 19 trans/μs on the LLC and 3.5 trans/μs on main memory to determine if a given processes is *heavy-* or *light-sharing*, depending on whether it will impact on the performance of its co-runners or not, respectively. The two deviation sources and the related parameters used in the *Perf&Fair* scheduler are further discussed in Appendix 1, available in the online supplemental material.

4 Background on Bandwidth-Aware Scheduling

Numerous schedulers have been proposed dealing with main memory or LLC bandwidth contention. Most of these proposals reduce contention by balancing the overall bandwidth utilization along the workload execution time. For this purpose, these approaches use a target main memory bandwidth that should be consumed on each quantum. Basically, these approaches differ among them on the way this target bandwidth is calculated.

Some works define the target bandwidth based on hardware parameters such as the peak main memory bandwidth [11]. Other works [8] use off-line information of the processes to determine the target bandwidth utilization. In [8], Xu et al. compute an *Ideal Average Bandwidth (IABW)* from the average bandwidth utilization, when running in isolation, of all the processes that compose the workload.

In a more recent work, Feliu et al. [12] define the *On-line Average Transaction Rate (OATR)* to be used as the target bandwidth for each quantum. This metric also pursues to evenly distribute the overall bandwidth utilization along the workload execution time. However, unlike the IABW, the OATR dynamically estimates the average on-line bandwidth utilization of the processes at run-time, and thus it does not require off-line information. The *Perf&Fair* scheduler proposed in this work follows this approach and uses the

OATR as the target bandwidth utilization for each quantum. See Section 5.2 for further details on the OATR calculation.

Regarding SMT cores, performance highly suffers due L1 bandwidth contention [13]. Therefore, the bandwidth-aware process allocation tries to reduce the intra-core interference by balancing the bandwidth utilization of the selected processes among L1 caches. Lower interference should not only improve performance but also fairness, since the interference causes wider performance differences among the running processes. Such L1-bandwidth aware process allocation has also been implemented in a process scheduler [12].

5 *PERF&FAIR* SCHEDULER

The proposed *Perf&Fair* scheduler confronts a twofold goal: enhancing performance and reducing unfairness. On the one hand, to improve performance the scheduler minimizes main memory and L1 cache bandwidth contention. Regarding main memory bandwidth, the scheduler tries to balance the overall memory requests of the processes along the mix execution time. Concerning L1 bandwidth, processes are allocated to cores balancing the overall L1 requests among all the L1 caches. On the other hand, to lessen unfairness the scheduler estimates the progress made by each process and prioritizes the processes with lower accumulated progress.

Two distinct scheduling modes are implemented in the *Perf&Fair* scheduler referred to as *IPC estimation -oriented* mode and *performance & fairness -oriented* mode. The former mode applies when any process needs to estimate its isolated IPC. The latter guides the scheduling to enhance performance and fairness, and applies when all the processes have a valid IPC estimate. As a good trade-off between IPC estimation accuracy and impact of number of estimation quanta on performance, IPC estimates for each process are kept valid for 40 quanta [14].

Performance counters play an essential role to implement the proposed scheduler and are used to dynamically compute IPC and bandwidth utilization of the processes. The IPC of the processes is used to estimate their progress. The main memory and L1 bandwidth utilization of the processes guide the process selection and process allocation, respectively. Finally, the main memory and LLC bandwidth utilization of the last executed quantum of the processes are also used to determine at runtime if they belong to the light- or heavy- sharing category as explained in Section 3.

Note that our aim is to propose a scheduling algorithm that addresses both performance and fairness simultaneously in multicores consisting of SMT cores. Nevertheless, if support to user-defined priorities (i.e., Linux *nice* priorities) is required, these can be determined with respect to the progress made by each process similar to how the Linux CFS scheduler uses the *nice value* to weight the proportion of processor a process is to receive [15]. In this context, a process with higher priority should progress faster than a process with a lower priority. Although it is possible to extend the proposed algorithm to allow a process to progress n% faster than others, where n depends on the nice value, the implementation and evaluation of this extension is out of the scope of this work.

Algorithm 1 presents the pseudocode of the proposed scheduler, which differentiates between both scheduling

Algorithm 1 Progress-Aware scheduler

1: Update IPC and bandwidth utilization for each process P run in the last quantum

PROCESS SELECTION

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

IPC-ESTIMATION MODE (Q_{length}=100ms)

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

PERFORMANCE & FAIRNESS MODE (Q_{length}=200ms)

11: Calculate $OATR = \frac{\sum_{p=0}^{N} Avg\_BW_{MM}^{p}}{N} \times$ #CPUs
12: Set $BW_{Remain} = OATR$, $CPU_{Remain} = \#CPUs$
13: Set $MaxP =$ Maximum progress $\forall$ P_X $\exists$ Process queue
14: **while** $CPU_{Remain} > 0$ **do**
15: $\forall$ P_i with Progress(P_i) + 1 $< MaxP$ **do**
16: Select the process P that maximizes
17: FITNESS(p) $= \frac{1}{\left| \frac{BW_{Remain}}{CPU_{Remain}} - BW_{MM}^{p} \right|}$
18: Update $BW_{Remain} -= BW_{MM}^{p}$, $CPU_{Remain}--$
19: **end while**
20: **end if**

PROCESS ALLOCATION

21: Allocate the threads that reserved an entire core to a core
22: Sort the remaining selected processes in ascending BW_{L1}
23: **while** there are unallocated processes **do**
24: Allocate the processes P_{head} and P_{tail} to the same core
25: **end while**

modes in the process selection: IPC estimation-oriented mode (lines 3 to 9) and performance- & fairness-oriented mode (lines 11 to 19), and the process allocation. Below, the main parts of the scheduler are discussed.

5.1 IPC Estimation-Oriented Mode

This mode (lines 2 to 9 of the algorithm) is triggered when a valid IPC estimate is required (line 2) for any process P. A low-contention scenario is scheduled to avoid intra-core interference and minimize the inter-core ones. The former interference is removed by allocating P to an entire core (line 3). The inter-core interference is minimized by only selecting light-sharing processes in the co-schedule.

If a process P is a light-sharing process (line 4), and there are other light-sharing processes with a relatively short time before expiring its current IPC estimate (half the number of quanta between standalone IPC estimates), then each of them is allocated to an individual core (line 6) [14]. This way allows multiple IPC estimates to be obtained during the same quantum, thus reducing the number of quanta devoted to IPC estimates.

After that, light-sharing processes are allocated to the remaining cores. In particular, as many light-sharing processes as available SMT hardware threads are co-scheduled (line 9). For the sake of fairness, the scheduler prioritizes the

light-sharing processes that have accumulated less progress. Notice that if not enough light-sharing processes are available in a given quantum, the exceeding hardware threads are kept free.

To increase performance, the estimation quanta length is set to 100ms (half the quantum length on Performance & Fairness -oriented mode). As discussed before, estimation quanta highly constrain the process selection, which strongly affects the performance. By halving the quantum length, the accuracy of the estimations is slightly reduced, which affects fairness, but important performance benefits can be achieved. Finally, notice that even for a quantum length of 100ms there is minor scheduling overhead.

5.2 Performance- & Fairness- Oriented Mode

In order to improve performance without sacrificing fairness, processes must be carefully selected. The main idea behind this mode consists in selecting the processes following a performance approach but preventing, as much as possible, unfairness from growing.

Regarding performance, to select the processes, the algorithm calculates first the On-line Average Transaction Rate (OATR) for the following quantum as the average main memory bandwidth (Avg_BW$_{MM}$) of the N processes of the workload multiplied by the number of hardware threads (#CPUs) supported by the experimental platform (i.e., the number of cores times the number of threads per core). The OATR represents the bandwidth that should be consumed at the next quantum in order to evenly distribute the memory requests along the workload execution time. A similar approach is used in other works to determine the optimal bandwidth consumption for each quantum [8], [9]. The OATR needs to be calculated for each quantum since the average bandwidth utilization of the processes is dynamically updated at the end of the quantum.

The OATR and the number of hardware threads are initially assigned to the variables BW$_{Remain}$ and CPU$_{Remain}$, respectively (line 12). These variables are iteratively updated inside the loop between lines 14 and 19 as processes are selected to be run in the next quantum. BW$_{Remain}$ represents the remaining bandwidth to be distributed among the processes that will be selected. Once a process P is selected to be run, its predicted main memory bandwidth utilization for the next quantum (BW$^P_{MM}$) is subtracted from the BW$_{Remain}$. BW$^P_{MM}$ is predicted to be equal to the bandwidth utilization measured in the last executed quantum of process P. CPU$_{Remain}$ represents the unallocated hardware threads and it is decremented by one once a process is selected.

To prevent unfairness from growing, the scheduler restricts the process selection (when it is possible) to the processes whose current progress is so low that if they were selected to run in the next quantum (Q_{i+1}), their progress after Q_{i+1} could not exceed the current (i.e., after quantum Q_i) maximum progress among all the processes of the mix. To do that, the algorithm determines the maximum progress $MaxP$ achieved among the running processes (line 13). Since the progress during a quantum is defined as IPC$_{co-runners}$ / IPC$_{alone}$, the maximum increase of progress that a process can experience in a quantum is 1. Based on this fact, only processes whose progress differs more than one unit from $MaxP$ are considered as schedulable at this point (line 15).

Among the processes that fulfill the previous condition, the fitness function determines which ones are finally selected (lines 16-17) attending to performance. The fitness function quantifies, for each process P, the gap between its predicted main memory bandwidth utilization for the next quantum (BW$^P_{MM}$) and the average bandwidth remaining per unallocated hardware thread (BW$_{Remain}$/CPU$_{Remain}$). This step is similar to the one performed in previous works focusing exclusively on performance [8], [9]. The process with the best fit maximizes the fitness function and it is selected to run during the following quantum, updating BW$_{Remain}$ and CPU$_{Remain}$ accordingly (line 18). After that, the algorithm proceeds with the next process selection iteration until CPU$_{Remain}$ is equal to 0 (line 14).

When the number of processes that fulfill the progress condition (line 15) is below the number of hardware threads, all these processes are directly selected to run for the following quantum, updating the BW$_{Remain}$ and CPU$_{Remain}$ variables. The remaining processes, until the number of hardware threads is reached, are selected using the fitness function (as explained before), but considering all the remaining processes regardless of their accumulated progress. These steps are not shown in the algorithm due to space constraints, but match lines 14-19 apart from the condition in line 15.

5.3 Process Allocation

Regarding process allocation, if the process selection has been performed by the IPC estimation oriented mode, some processes will require an entire core for themselves. Thus, the first step of the process allocation assigns these processes to entire cores (line 21). After that, the remaining processes are sorted in a list in ascending L1 bandwidth order (line 22). Then, the processes placed at the head and the tail of the list are removed from the list and allocated together to the same core. This action is performed iteratively until the list is empty (lines 23-25).

6 Evaluation Methodology

6.1 Scheduler Implementation

To evaluate the effectiveness of the *Perf&Fair* scheduler, we compare its fairness and performance to those achieved by the Linux OS scheduler, a state-of-the-art performance-oriented scheduler [12] (*Perf*), and a state-of-the-art fairness-oriented scheduler [14] (*Fair*). The studied algorithms are 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 using the events *Offcore_Response_0.Any_Data.Local_DRAM*, *Offcore_Response_0.Any_Data.Local_Cache*, and *Perf_Count_HW_Cache_L1D.Access*, respectively. 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. Quantum length is set to 200ms, except for IPC-estimation oriented quanta in the *Perf&Fair* scheduler, which are set to 100ms.

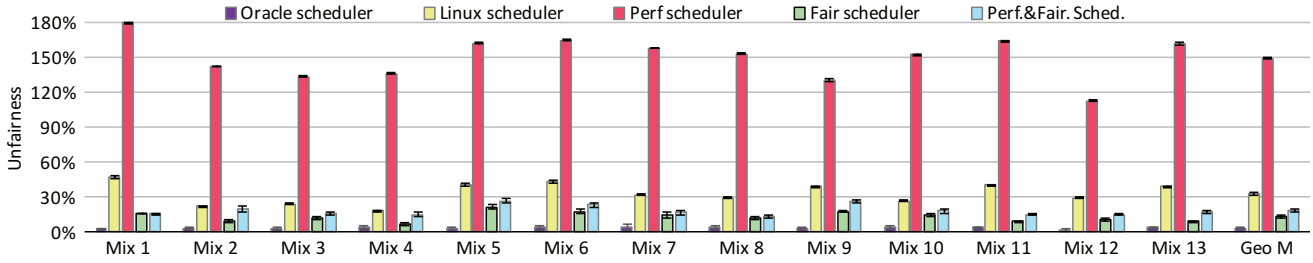

Fig. 1: Unfairness (lower is better) with different scheduling policies, including 95% confidence intervals.

Linux schedules are also evaluated with the user-level scheduler to monitor the number of instructions each process executes (see Appendix 2, available in the online supplemental material). To *mimic* the Linux behavior with this user-level scheduler, all the processes are allowed to run each quantum on any core, so the Linux kernel scheduler determines the actual co-schedule (both process selection and process allocation) [8], [9].

The overhead arising from the algorithm implementation is negligible considering the quantum lengths 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 0.1ms. Note that it is below 0.1% of the quantum length.

A set of thirteen mixes composed of twenty-four SPEC CPU2006 benchmarks has been designed to evaluate the proposed algorithm. Mixes have been sorted according to their average main memory bandwidth consumption (BW_{MM}). Further details of the mix design, including a table of their composition, are presented in Appendix 2.

Since fairness can be achieved at the cost of performance, it should not be evaluated in isolation but performance metrics should also be considered. In this work we use the turnaround time [8], [9] of the mixes as performance indicator. This metric measures the elapsed time since the workload is launched to execution until the last process of the workload is completed. Regarding fairness, we use the *unfairness* metric [7], [16], [17], which is defined as the maximum slowdown divided by the lowest slowdown across all the processes of the workload. Further details of the unfairness metric and its calculation are discussed in Appendix 2.

7 Experimental Evaluation

This section evaluates the fairness and performance achieved by the following schedulers:

- **Linux**: the default Linux Completely Fair Scheduler (CFS) in our experimental platform.
- ***Perf&Fair***: our proposed algorithm.
- ***Perf***: a performance-oriented bandwidth-aware scheduler [12] that only deals with main memory and L1 bandwidth contention to improve performance. Regarding main memory contention, Perf selects each quantum those processes that maximize the fitness function (see Section 5.2) without taking into account any progress consideration. Process starvation is avoided by selecting each quantum the process that has not been scheduled for longer. Regarding L1 bandwidth contention, Perf allocates the selected processes to hardware threads balancing the L1 utilization among cores as explained in Section 5.3.
- ***Fair***: a progress-aware scheduler [14] designed exclusively to maximize fairness. This algorithm schedules the processes periodically in low contention scenarios to estimate their isolated performance, which is used to compute their progress. Then, to maximize fairness, the processes with lower accumulated progress are prioritized. It also deals with L1-bandwidth contention to reduce the interference, which can also provide performance benefits.
- ***Oracle***: the *Perf&Fair* algorithm enhanced with offline information. This enhancement uses standalone IPC traces to compute the progress of the processes and the *Ideal Average Bandwidth* (IABW) utilization [8], which it is used instead of the OATR in the algorithm. The IABW is conceptually equivalent to the OATR, but it is obtained using preliminary knowledge of isolated bandwidth requirements of the processes, avoiding any estimate.

7.1 Fairness

Figure 1 depicts the unfairness, in percentage, presented by the Linux, *Perf*, *Fair*, *Perf&Fair*, and *Oracle* schedulers across the studied mixes. The *Perf* scheduler reaches extremely high levels of unfairness (geometric mean by 149%), which means that the last process finishes its execution by 2.5× later than the first process. Notice that the rule included in *Perf* to avoid starvation is not able to keep unfairness at a reasonable level.

The Linux scheduler is the second one with highest unfairness. Under the Linux scheduler six mixes present an unfairness around 40% and a geometric mean by 33%. Although much lower than that shown by the *Perf* scheduler, this level of unfairness still seems high and might be inappropriate in some systems. In contrast, executions with the *Fair* scheduler do not surpass an unfairness of 20%, with the only exception of mix 5. The *Fair* scheduler exhibits an unfairness of 13.5%, approximately 2.5× lower than the unfairness shown by Linux.

The *Perf&Fair* scheduler shows similar unfairness as that of the *Fair* scheduler. Unfairness only surpasses 20% in four mixes and the geometric mean is by 18.5%. The achieved unfairness makes a big difference with respect to that achieved by Linux, where all the mixes surpass an unfairness of 20% (except mix 4). In addition, the geometric mean of the unfairness is reduced from 33% to 18.5%.

Finally, by using offline traces of the standalone performance of the processes, the *Oracle* scheduler performs nearly completely fair, reaching an average unfairness by

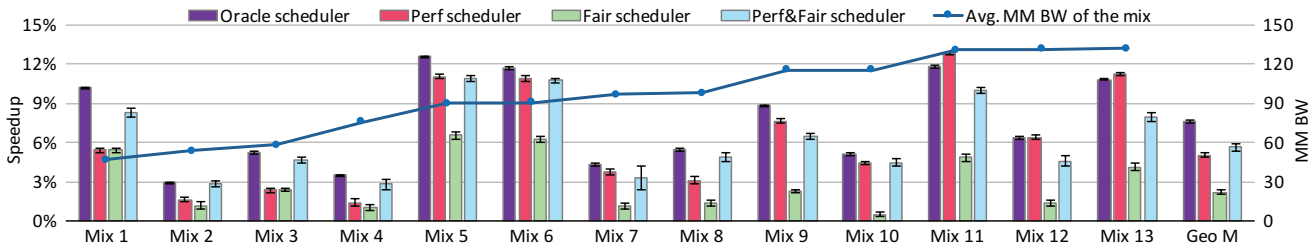

Fig. 2: Speedup of the turnaround time with respect to Linux. The bars show the speedup achieved with different scheduling policies, including 95% confidence intervals. The line shows the average main memory bandwidth of the mixes.

3.5%. The results show that *Perf&Fair*, without using offline information, offers a fairness that is close to that obtained with exact knowledge of the progress made by each process. The results of the *Oracle* scheduler also show that despite not being exclusively focused on improving fairness, the *Perf&Fair* scheduling algorithm can perform nearly completely fair if the progress estimates are completely accurate. In other words, addressing performance additionally to fairness does not affect the optimal unfairness that the *Perf&Fair* scheduler can achieve.

7.2 Performance

Figure 2 presents the speedup of the turnaround time achieved by the *Perf*, *Fair*, *Perf&Fair*, and *Oracle* scheduling algorithms over the Linux scheduler. At a first glance, it can be observed that, in spite of reaching an unfairness close to the *Fair* scheduler, the *Perf&Fair* scheduler achieves speedups closer to the *Perf* scheduler than to the *Fair* scheduler. In fact, it enhances the speedups achieved with the *Fair* scheduler in all the mixes. Considering all the evaluated mixes, the *Perf&Fair* scheduler reaches the highest geometric mean of speedup (5.6%), followed by the *Perf* scheduler (5%), and the *Fair* scheduler (2.2%). The figure also plots the average BW_{MM} of the mixes (solid line and secondary y axis), which helps understand the achieved results. Note that the studied mixes are sorted in increasing average BW_{MM} order. Mixes can be divided in three main groups which present different behavior according to their average BW_{MM}.

When the average BW_{MM} is relatively low (below 80 trans/usec), though still significant, bandwidth contention and progress unbalancing similarly affect the turnaround time of the mixes. This is because at the end of an unbalanced execution the number of available processes is less than the number of hardware threads during a significant number of quanta, which penalizes the turnaround time. In addition, since the bandwidth contention is not as high as in other workloads, the benefits of a better main memory bandwidth management decrease and can be canceled due to highly unbalanced executions. In these scenarios, a fairer scheduler, through a better progress balancing, can reduce the number of quanta where there are less available processes than hardware threads and reduce the turnaround time. In fact, it can be observed that the *Perf* and *Fair* schedulers reach similar speedup, despite they schedule processes following a completely different strategy. Moreover, *Perf&Fair* effectively combines both performance and fairness approaches and, by concurrently mitigating bandwidth contention while keeping unfairness under control, it improves the speedups achieved by both *Perf* and *Fair*.

As the average required memory bandwidth of the mixes grows, bandwidth contention becomes a major performance limitator, which translates in larger turnaround times. When the bandwidth falls in between 80 and 120 trans/usec, the *Perf* scheduler benefits enough from the bandwidth contention to improve performance over the *Fair* scheduler. In this scenario, *Perf&Fair* still reaches speedups closely resembling *Perf*, since it is still able to address bandwidth contention while keeping a good progress balancing among the processes.

In the most memory-bounded mixes studied, with above 120 trans/usec, the *Perf* scheduler, which exclusively addresses bandwidth contention improves performance over the *Fair* scheduler. In this case, *Perf&Fair* widely improves the results of both *Fair* and Linux, but it is not able to reach speedups as high as those achieved with *Perf* because it also deals with unfairness. Therefore, it cannot devote all selected processes to maximize performance.

Regarding the *Oracle* scheduler, by using offline traces it further enhances the performance of *Perf&Fair*, despite the benefits are not too large on some workloads (e.g., mix 2 and mix 3). The use of traces also allows the *Oracle* scheduler to improve *Perf* in workloads with low and medium main memory bandwidth utilization, since it achieves a better progress balancing. In the workloads with he highest main memory bandwidth utilization, *Oracle* reaches speedups very close to *Perf* but slightly below, since the former scheduler is partially constrained by fairness requirements.

Finally, it should be emphasized that the *Perf&Fair* scheduler addresses both performance and fairness without requiring from offline traces. Thus, if both metrics are considered together, this scheduler is the one that behaves more satisfactorily. Moreover, the algorithm is flexible enough by design and can be modified to provide different tradeoffs between performance and fairness (see Appendix 3, available in the online supplemental material).

7.3 Processes Completion in a Mix

To provide insights on the obtained turnaround time and unfairness results, we focus the analysis on mix 9, where the schedulers present widely different results.

Figure 3 presents how the number of processes of mix 9 evolve over time when this mix is executed under the studied schedulers. The plot starts at quantum 700, where no process has yet finished, and shows how the execution of the processes is being completed. The time at which the last process of the 24-task mix finishes its execution determines its turnaround time. The *Perf* scheduler shows the shortest turnaround time, closely followed by the *Perf&Fair* scheduler, then the *Fair* scheduler, and finally the Linux scheduler,

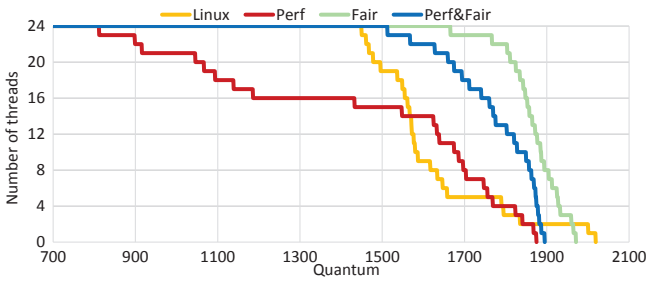

Fig. 3: Number of remaining processes along the execution of mix 9 with the studied schedulers.

which shows the largest turnaround time. On the other side, unfairness is determined as the ratio between the time at which the first and the last processes of the mix complete their execution. As observed, the *Fair* scheduler achieves the lowest unfairness, followed by the *Perf&Fair*, Linux and *Perf* schedulers.

The figure also illustrates the importance of fairness, or progress balancing, on the turnaround time of the mix. For instance, Linux is the first scheduler to complete the execution of 12 processes, but the last to complete the execution of the whole mix. This means that in this execution Linux takes the system to a low loaded state (i.e., with less applications than hardware threads) faster than other schedulers (by quantum 1600). However, this behavior is at the cost of system fairness, since Linux takes longer time to complete the last five processes of the workload, even if at this point each process is running alone on a different core. Another observation is that the *Perf* scheduler completes the first process as soon as quantum 812, which yields it to the highest unfairness. Finally, regarding the *Perf&Fair* curve it is interesting to observe how, despite having a turnaround time close to the *Perf* scheduler, it presents a process completion curve resembling that of the *Fair* scheduler.

8 Related Work

Contention in shared resources has been addressed in scheduling algorithms, but an important piece of this work in the past [8], [9], [11], [18], [19], [20] mainly focuses on performance without taking fairness into account. Some works address main memory bandwidth contention [8], [11]. Antonopoulos et al. [11] propose to schedule processes trying to match the peak memory bus bandwidth. In a more recent approach, Xu et al. [8] observe that contention exists below the peak memory bandwidth due to irregular access patterns and propose to distribute the overall main memory requests over the workload execution time.

Other work focuses on LLC contention.. Tang et al. [18] study the impact of sharing memory resources and find that improperly sharing the LLC can degrade performance, while Zhuravlev et al. [19] propose a scheduling algorithm that, among other resources, addresses contention for LLC space. More recently, Feliu et al. [9] present a scheduling algorithm that deals with bandwidth contention along multiple levels of the memory hierarchy. A complete survey on scheduling techniques to address contention in the shared CMP resources was presented by Zhuravlev et al. [20].

Scheduling has also been studied in SMT processors.Parekh et al. [21] proposed some of the first thread-sensitive scheduling policies to improve the performance of SMT processors based on metrics such as the IPC, and the miss rate on the L1 data cache, the L2 cache, and the data TLB. Other works have studied job symbiosis as a way to rise performance by co-scheduling jobs with *compatible* demands on the shared resources. In recent approaches, Eyerman et al. [22], [23] and Feliu et al. [24] predict how the interference among processes affects the performance of different co-schedules without actually running them, which is used to select the co-schedule with highest performance on each quantum. With the same goal, but focused on L1 bandwidth, Feliu et al. [13] propose a process allocation policy to balance the overall L1 requests across all the L1 caches of the processor. A different approach is followed by Saez et al. [25], who propose a non-work-conserving scheduler that greatly speedups critical threads while still achieving slight throughput improvements on ST and SMT multicores. Also targeting SMT multicores but focused on multithreaded applications, Funston et al. [26] propose an SMT metric to select the optimal number of threads per core depending on the instruction mix of the application.

Regarding fairness, it has been addressed in several works from a shared resource perspective, trying to provide fair sharing in a given resource. Some of them focus on uncore memory resources, and particularly on the memory controller [2], [5], to improve the system fairness. Mutlu et al. [2] propose a memory access scheduler that balances the DRAM-related slowdown experienced by the co-scheduled processes, and a similar approach is followed by Nesbit et al. [5] using concepts from network queuing to design a fair queuing memory system. Finally, Ebrahimi et al. [16] 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 [27], [28] or cache partitioning mechanisms [6], [29].

Unfortunately, fairly sharing a single resource or a set of them does not provide system fairness. Thus, other authors aim to provide fairness by focusing on process scheduling. Fedorova et al. [30] present a scheduler that targets shared-cache contention using resource performance, while Xu et al. [7] mainly target main memory contention and focus on overall system fairness with a scheduler that monitors the progress of the processes at runtime. Dealing with SMT multicores, the first approach to attack unfairness was proposed by Parekh et al. [21]. However, it is based on the number of quanta each process is run instead on its actual progress during these quanta. In a more recent work, Feliu et al [12] propose a scheduling algorithm that estimates the progress experienced by the processes and gives priority to the processes with lower progress to reduce unfairness.

Apart from scheduling algorithms, techniques such as the CPU accounting mechanisms for multicore and SMT multicore processors [31], [32] can improve the accuracy of the progress estimates, which enables a better control of system fairness.

The discussed work tends to focus either on performance or fairness when scheduling processes. Unlike these approaches, the *Perf&Fair* scheduler deals with both of them simultaneously in SMT multicores.

9 Conclusions

While existing scheduling algorithms focus either on performance or fairness, this work presents the *Perf&Fair* scheduler for SMT multicores, aimed at providing the best of both worlds, by simultaneously addressing performance and fairness in multiprogrammed workloads. The design of such a kind of algorithms is a major challenge since improving a given factor can easily damage the other one.

To deal with performance, the proposed scheduler balances the bandwidth consumption among the available resources and along the execution time of the workload. To reduce unfairness, the scheduler estimates the progress made by the processes, and gives priority to the processes with lower accumulated progress.

Experimental results obtained in a Intel Xeon E5645 with six dual-threaded SMT cores show that the *Perf&Fair* scheduler accomplishes its two-fold goal. Regarding performance, the *Perf&Fair* scheduler achieves speedups of the turnaround time of the mixes that slightly enhance the performance of a state-of-the-art performance-aware scheduler, with the only exception of extreme bandwidth-contention workloads. Across the set of evaluated workloads the speedup of the *Perf&Fair* and the performance-aware schedulers over Linux are, on average, by 5.6% and 5%, respectively. The key is that such a level of performance is achieved while unfairness is reduced from a geometric mean of 149% and 33% of the performance-aware and Linux schedulers, respectively, to only 18.5% in the proposed *Perf&Fair* scheduler.

Acknowledgments

We thank the anonymous reviewers for their constructive and insightful feedback. This work was supported in part by the Spanish Ministerio de Economía y Competitividad (MINECO) and Plan E funds, under grants TIN2015-66972-C5-1-R and TIN2014-62246-EXP, 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 *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchit.*, 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 Trans. Comput.*, 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 *Proc. 16th USENIX Security Symp.*, 2007, pp. 18:1–18:18.

[5] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2006, pp. 208–222.

[6] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proc. 13th Int. Conf. Parallel Archit. Compilation Tech.*, 2004, pp. 111–122.

[7] D. Xu, C. Wu, P.-C. Yew, J. Li, and Z. Wang, "Providing fairness on shared-memory multiprocessors via process scheduling," in *Proc. 12th ACM SIGMETRICS/PERFORMANCE Joint Int. Conf. on Measurement and Modeling of Comput. Syst.*, 2012, pp. 295–306.

[8] D. Xu, C. Wu, and P.-C. Yew, "On mitigating memory bandwidth contention through bandwidth-aware scheduling," in *Proc. 19th Int. Conf Parallel Archit. Compilation Tech.*, 2010, pp. 237–248.

[9] J. Feliu, S. Petit, J. Sahuquillo, and J. Duato, "Cache-hierarchy contention aware scheduling in CMPs," in *IEEE Trans. Parallel and Distrib. Syst.*, vol. 25, no. 3, March 2014, pp. 581–590.

[10] S. Eranian, "What can performance counters do for memory subsystem analysis?" in *Proc. ACM SIGPLAN Workshop Memory Syst. Perform. Correctness*, 2008, pp. 26–30.

[11] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou, "Realistic workload scheduling policies for taming the memory bandwidth bottleneck of SMPs," in *Proc. 11th Int. Conf. on High Perform. Comput.*, 2004, pp. 286–296.

[12] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "Bandwidth-aware online scheduling in SMT multicores," *IEEE Trans. Comput.*, vol. 65, no. 2, pp. 422–434, Feb. 2016.

[13] ——, "L1-bandwidth aware thread allocation in multicore SMT processors," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Tech.*, 2013, pp. 123–132.

[14] ——, "Addressing fairness in SMT multicores with a progress-aware scheduler," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, 2015, pp. 187–196.

[15] R. Love, *Linux kernel development*. Pearson Education, 2010.

[16] 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 *Proc. 15th Ed. Archit. Support Program. Languages Operating Syst.*, 2010, pp. 335–346.

[17] 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 *Proc. 19th Int. Symp. High-Performance Comput. Archit.*, 2013, pp. 107–118.

[18] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M.-L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *Proc. 38th Annu. Int. Symp. Comput. Archit.*, 2011.

[19] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proc. 15th Ed. Archit. Support Program. Languages Operating Syst.*, 2010, pp. 129–142.

[20] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 4:1–4:28, Dec. 2012.

[21] S. Parekh, S. Eggers, H. Levy, and J. Lo, "Thread-sensitive scheduling for SMT processors," 2002.

[22] S. Eyerman and L. Eeckhout, "Probabilistic job symbiosis modeling for SMT processor scheduling," in *Proc. 15th Ed. Archit. Support Program. Languages Operating Syst.*, 2010, pp. 91–102.

[23] S. Eyerman, P. Michaud, and W. Rogiest, "Revisiting symbiotic job scheduling," in *IEEE Int. Symp. Perform. Analysis Syst. Software*, 2015, pp. 124–134.

[24] J. Feliu, S. Eyerman, J. Sahuquillo, and S. Petit, "Symbiotic Job Scheduling on the IBM POWER8," in *Proc. IEEE 22nd Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 669–680.

[25] J. C. Sez, J. I. Gomez, and M. Prieto, "Improving priority enforcement via non-work-conserving scheduling," in *Proc. 37th Int. Conf. Parallel Process.*, 2008, pp. 99–106.

[26] J. R. Funston, K. E. Maghraoui, J. Jann, P. Pattnaik, and A. Fedorova, "An SMT-selection metric to improve multithreaded applications' performance," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, 2012, pp. 1388–1399.

[27] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Thoughput and Fairness in SMT Processors," in *IEEE Int. Symp. Perform. Analysis Syst. Software*, 2001, pp. 164–171.

[28] S. Eyerman and L. Eeckhout, "A memory-level parallelism aware fetch policy for SMT processors," in *Proc. IEEE 13th Int. Symp. High Perform. Comput. Archit.*, Feb 2007, pp. 240–249.

[29] G. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proc. IEEE 8th Int. Symp. High Perform. Comput. Archit.*, 2002, pp. 117–128.

[30] A. Fedorova, M. Seltzer, and M. D. Smith, "Improving performance isolation on Chip multiprocessors via an operating system scheduler," in *Proc. 16th Int. Conf. Parallel Archit. Compilation Tech.*, 2007, pp. 25–38.

[31] C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero, "CPU accounting for Multicore Processors," *IEEE Trans. on Comput.*, vol. 61, no. 2, pp. 251–264, Feb 2012.

[32] C. Luque, M. Moreto, F. J. Cazorla, and M. Valero, "Fair CPU time accounting in CMP+SMT processors," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 50:1–50:25, Jan. 2013.