Bandwidth-Aware On-Line Scheduling in SMT Multicores

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

Abstract—The memory hierarchy plays a critical role on the performance of current chip multiprocessors. Main memory is shared by all the running processes, which can cause important bandwidth contention. In addition, when the processor implements SMT cores, the L1 bandwidth becomes shared among the threads running on each core. In such a case, bandwidth-aware schedulers emerge as an interesting approach to mitigate the contention. This work investigates the performance degradation that the processes suffer due to memory bandwidth constraints. Experiments show that main memory and L1 bandwidth contention negatively impact the process performance; in both cases, performance degradation can grow up to 40% for some of applications. To deal with contention, we devise a scheduling algorithm that consists of two policies guided by the bandwidth consumption gathered at runtime. The process selection policy balances the number of memory requests over the execution time to address main memory bandwidth contention. The process allocation policy tackles L1 bandwidth contention by balancing the L1 accesses among the L1 caches. The proposal is evaluated on a Xeon E5645 platform using a wide set of multiprogrammed workloads, achieving performance benefits up to 6.7% with respect to the Linux scheduler.

Index Terms—bandwidth-aware scheduling; process selection; process allocation; L1-bandwidth, bandwidth contention; SMT

1 Introduction

Simultaneous multithreading (SMT) processors exploit both instruction-level and thread-level parallelism by issuing instructions from different threads in the same cycle. Thread-level parallelism increases the chance of having instructions ready to be issued thus reducing the vertical waste at the issue stage [1]. Because of different threads can launch instructions in the same cycle, threads are continuously sharing processor resources. Thus, the performance of SMT cores strongly depends on how resources are shared among threads.

The subset of processor resources being shared depends on the actual SMT implementation but typically includes, among others, functional and arithmetic units, instruction queues, renaming registers and first-level caches. If at any point of the execution time, the demand for a given resource exceeds its capability, the performance can be seriously affected. Thus, smart thread to core (t2c) mapping policies can help alleviate the contention in shared resources in current multicore multithreaded processors. On the contrary, a *naive* policy could stress the contention on some resource so creating a new performance bottleneck.

A critical resource in any current chip multiprocessor (CMP) is the main memory bandwidth, which is shared among all the processor cores. For a given system, the higher the number of cores the higher the potential contention due to main memory bandwidth constraints. Climbing the memory hierarchy, LLC caches (and caches of higher levels) are also typically shared by a subset or all the cores; thus, bandwidth contention can rise at different points of the memory hierarchy. Main memory [2], [3] and LLC bandwidth [4], [5], [6] have been addressed in recent research work that illustrates the potential performance improvements that bandwidth-aware scheduling policies can offer by providing a better sharing of the memory hierarchy resources.

In summary, research work on CMPs has focused on scheduling strategies to tackle bandwidth contention, and research work based on SMT processors has concentrated on core management policies for shared resources. However, to the best of our knowledge, L1 bandwidth, which is private to cores in CMP systems but shared to threads in SMT cores, has not been addressed yet neither in scheduling nor resource sharing strategies.

This paper proposes a scheduling algorithm for multicore SMT processors that deals with memory bandwidth at different points of the memory hierarchy. The proposed scheduler consists of two main policies, process selection and process allocation.

The process selection step is based on the main memory bandwidth the processes consume, which is gathered at runtime with performance counters. This policy contributes to enhance the performance by choosing an adequate subset of processes of the workload to be run during each quantum. Previous work [3], [6] dealing with bandwidth contention requires information from prior executions to feed the scheduler, which makes this approach impractical on real systems. This work also devises a process selection policy with the aim of fairly balancing memory bandwidth across the workload execution time. However, unlike previous work, the proposed policy, referred to as *On-line* process selection policy, does not require any prior information on the

- *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*

processes.

Once the processes to be run have been selected, the process allocation strategy determines the target core for each process. This work shows that, i) there is a strong connection between the performance of SMT cores and the L1 bandwidth they consume and, ii) the bandwidth a process consumes (in a 2-thread SMT) is strongly related to that of its co-runner. These findings suggest that the more balanced the L1 bandwidth consumption, the higher the performance. For this purpose, the devised process allocation policy gathers L1 bandwidth requirements of individual threads at runtime and, dynamically updates the process to core mappings.

Finally, this work introduces the *consumed slots* plots, as a visual and intuitive approach to analyze the scheduler performance. As an example, Figure 1 shows, for each quantum, the number of consumed slots (i.e. the number of threads running at each OS quantum) during the execution of a workload with the Linux scheduler and the Bandwidth-Aware On-line Scheduler (BAOS) proposed. The figure highlights that scheduling brings performance benefits in two main ways. First, the time required to complete the whole multiprogrammed workload is shortened. Second, performance is also improved by finishing earlier any thread of the workload as it is shown in the shaded area enclosed between the consumed slots plots of both schedulers.

Experimental results on a Xeon E5645 processor show that the BAOS scheduler can significantly improve the performance with respect to the Linux scheduler. The achieved speedup varies depending on the evaluated metric and the workload but it grows up to 6.7%, with an geometric mean of 4.6% for the evaluated mixes using the average IPC.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the experimental platform. Section 4 analyzes the effects of L1 and main memory bandwidth on performance. Section 5 presents the scheduler. The evaluation methodology is described in Section 6, and the performance of the proposal is evaluated in Section 7. Finally, Section 8 presents some concluding remarks.

2 Related work

Some preliminary work on scheduling has focused on main memory bandwidth contention. Antonopoulos et al. [2] proposed to schedule processes so that their bandwidth consumption matches the peak memory bus bandwidth. In contrast, Xu et al. [3] proved that contention can rise even when memory bandwidth requirements are below the peak bandwidth due to irregular access patterns and thus, they distribute the memory accesses over the workload execution to minimize contention.

Other work has focused on cache contention. Eklov et al. [7] presented a method for measuring application performance and main memory bandwidth utilization as a function of the available shared cache capacity.

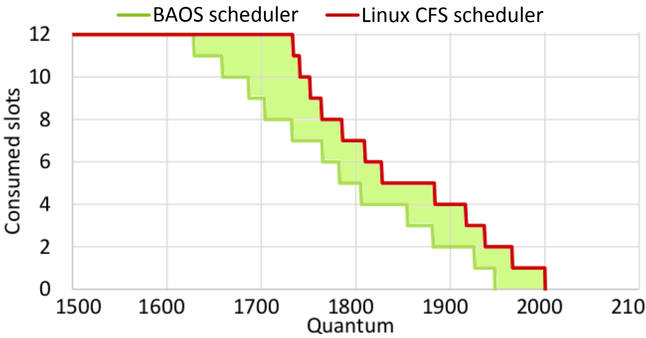


Fig. 1: Evolution of consumed slots during the execution of a sample workload.

Similarly, Casas et al. [8] presented a methodology to predict the performance of an application when the available bandwidth and space through the memory hierarchy are reduced. Some scheduling algorithms have also been designed addressing cache contention. Tang et al. [4] studied the impact of sharing memory resources and found that improperly sharing the LLC can degrade the performance, while Zhuravlev et al. [5] proposed a scheduling algorithm that, among other resources, addresses contention due to LLC space. In a similar way, Knauerhase et al. [9] devised a scheduler that profiles task execution with hardware counters to provide co-schedules that reduce cache interference. Fedorova et al. [10] proposed a *cache-fair* scheduling algorithm that gives more execution time to those processes more affected by unbalanced cache sharing. More recent scheduling strategies consider several levels of the memory hierarchy. Feliu et al. [6], [11] addressed bandwidth contention along the memory hierarchy of CMPs.

The predominant approach in current processors combines multicore and multithreading. In this kind of processors, thread allocation plays a key role on performance due to the multiple and heterogeneous levels of resource sharing. Settle et al. [12] proposed a thread scheduler on a single-core SMT processor, which used *activity vectors* to determine the pairs of threads with the lowest performance degradation when running simultaneously. More recently, Eyerman et al. [13] studied job symbiosis and proposed a model to predict whether jobs create positive or negative symbiosis when co-scheduled without actually running the co-schedule. Concerning SMT multicores, Čakarević et al. [14] characterized different types of resource sharing in an UltraSPARC T2 processor and improved the execution of multithreaded applications with a resource sharing aware scheduler. Acosta et al. [15] showed that processor throughput is highly dependent on thread allocation and proposed an allocation policy that combines computation and memory bounded processes in each core.

Some resource partitioning proposals also deal with bandwidth contention. Moretó et al. [16] partition the LLC of CMPs to increase memory level parallelism and reduce workload imbalance, and cache partitioning algorithms like SHARP [17] and PriSM [18] manage LLC cache sharing in CMPs using formal control and probability theories, respectively. Focusing more in bandwidth than in cache space, Nesbit et al. [19] propose a resource sharing mechanism that provides QoS to concurrent running processes. In particular, authors present an arbiter that guarantees a minimum bandwidth to the processes to provide QoS. A similar approach is followed by Colmenares et al. [20], who implement the Adaptive Resource Centric Computing (ARCC) in the Tessellation OS. Using ARCC, resources can be distributed to the processes providing performance isolation and predictability. Unfortunately both proposals focus on single-threaded processors and L1-bandwidth sharing among simultaneous threads is not addressed.

3 Experimental platform

Experiments have been performed on an Intel Xeon E5645 processor, composed of six dual-thread SMT cores. Each core includes two levels of private caches, a 32KB L1 and a 256KB L2. A third-level 12 MB cache is shared by the private L2 caches. 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 [21] and collects, for each running thread, the processor cycles and executed instructions, as well as the number of requests to L1 caches and to the main memory. The scheduler gathers these values at runtime to obtain bandwidth information, which is used 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 200 seconds in stand-alone execution. Benchmarks with shorter or longer execution time are relaunched or killed, respectively, to run exactly that amount of instructions.

4 Performance degradation analysis

4.1 Performance Degradation due to Main Memory Bandwidth Contention

The goal of this analysis is not to perform an in-depth study of the performance degradation caused by main memory bandwidth contention, but to provide an overall overview of how this contention affects the performance of the processes with the aim of motivating the use of a main memory bandwidth-aware process selection. A deeper analysis of the effects of bandwidth contention on performance can be found in [11].

To check the performance degradation caused by main memory bandwidth contention we designed a microbenchmark that presents a main memory transaction rate (TR$_{MM}$) of 55 trans/μs in stand-alone execution. The microbenchmark is designed to minimize the cache space contention, distributing the accesses among all the cache sets, so that the measured performance degradation is caused by bandwidth contention [11]. The performance degradation that each benchmark suffers is analyzed when it runs concurrently with one and five instances of the designed microbenchmark, respectively. The former scenario evaluates a situation with only one microbenchmark, which emulates one memory-bounded co-runner[1]. The latter evaluates the scenario with highest main memory bandwidth contention. In this case, the system executes six processes (one on each core): the studied benchmark and five instances of the microbenchmark. Because of the high TR$_{MM}$ of the designed microbenchmark, we guarantee that these five co-runners are enough to entirely consume the available main memory bandwidth.

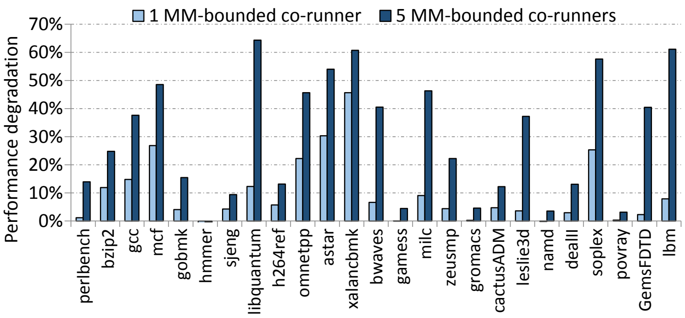


Fig. 2: IPC degradation due to main memory bandwidth contention.

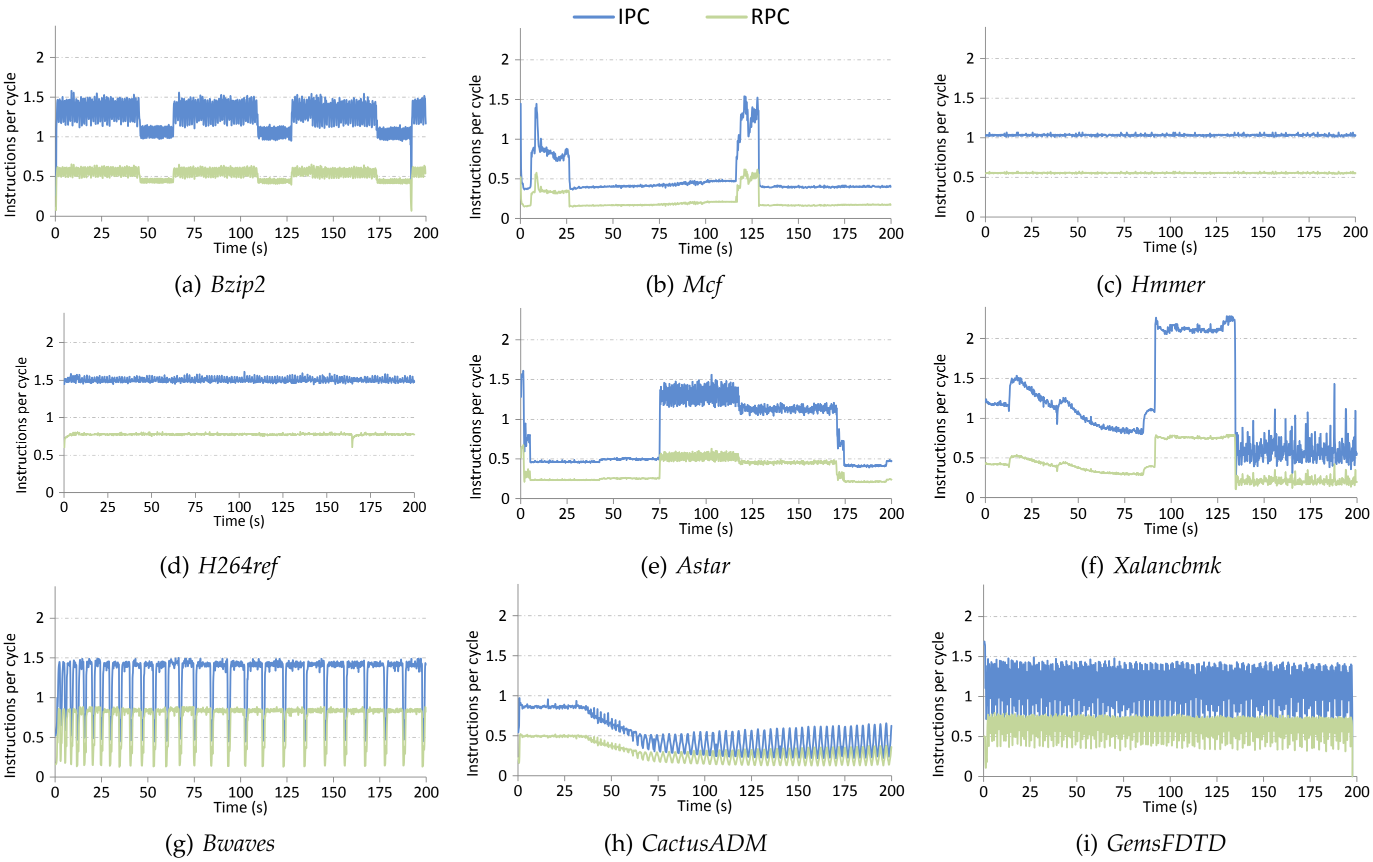
Figure 2 shows the performance degradation of the benchmarks in the devised experiment. When running with only one memory-bounded co-runner, the highest performance degradation observed is around 45% in *xalancbmk*, but it is smaller than 10% in half of the benchmarks. However, when running with five memory-bounded co-runners, performance degradation increases dramatically to the extent that half of the benchmarks suffer a degradation above 30% and five of them exceed 50%. Such degradations show the convenience of using a process selection based on the main memory bandwidth requirements of the processes.

4.2 Effects of L1 Bandwidth on Performance

Current microprocessors usually deploy a cache hierarchy organized in two or three levels of caches. The first-level cache, the closest one to the processor, is the most frequently accessed one. Consequently, L1 caches are critical for performance and thus, they are designed to provide fast access and high bandwidth.

This section analyzes the relationship between L1 bandwidth consumption and processor performance. First, we present a summary of the analysis of the behavior in stand-alone execution [22]. Then, we study

[1] The term co-runner is used to refer to the processes running concurrently that share the available bandwidth at a contention point. Regarding main memory, the co-runners of a process are all the processes running concurrently, while when addressing L1 bandwidth, the co-runner is the process running on the same core.



(a) *Bzip2*

(b) *Mcf*

(c) *Hmmer*

(d) *H264ref*

(e) *Astar*

(f) *Xalancbmk*

(g) *Bwaves*

(h) *CactusADM*

(i) *GemsFDTD*

Fig. 3: IPC and RPC evolution over time for a set of benchmarks.

how the interaction between two co-runners running in the same core affects their achieved performance and L1 bandwidth consumption.

4.2.1 Stand-Alone Execution

Figure 3 depicts the results of the execution of several quanta for a representative subset of benchmarks. Each plot presents, for a given benchmark, the instructions per cycle (IPC) and the number of instructions that perform a L1 data cache read per cycle (RPC)[2]. Thus, RPC is related with L1 bandwidth consumption.

The presented plots help detect the strong connection between RPC and IPC metrics. As observed, both metrics show an almost identical shape during the entire execution time across all the benchmarks. The metrics follow the same trend (rises and drops) in a synchronized way. This means that high (or low) IPC is typically correlated with high (or low) L1 bandwidth consumption. Note that, as soon as the L1 bandwidth starts to decrease (or increase), the performance of the process follows the same trend. Therefore, a key issue for performance is to schedule processes to cores with the aim of maximizing L1 bandwidth consumption.

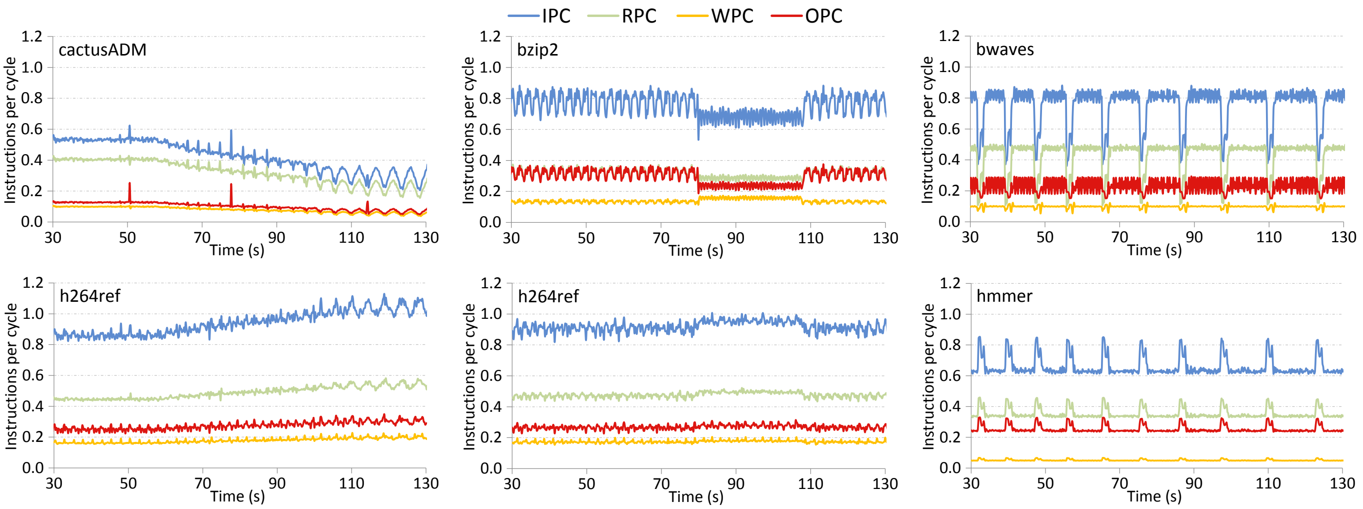
4.2.2 Analyzing Interferences Between Co-Runners

While current microprocessors implement LLC caches, which are shared by a subset or all the cores, L1 caches are designed to be private to cores. In case of single-threaded cores, all available L1 bandwidth is devoted to the same process. In such a system, processes do not compete for L1 bandwidth. In contrast, in current SMT processors, those threads running concurrently on the same core share the L1 cache. Since, as shown above, the IPC of a process depends on the L1 bandwidth it uses, its performance suffers when several threads run on the same SMT core because they compete for the available L1 bandwidth.

This section analyzes how sharing the L1 bandwidth limits the thread performance. To this end, multiple experiments running a couple of benchmarks on a single dual-threaded core were performed. To clearly show the impact of limited bandwidth on performance, the L1 bandwidth utilization of the benchmarks that run concurrently must fulfill two key characteristics. First, at least one benchmark with high L1 bandwidth requirements must be included to accentuate the impact of the contention on performance. Second, at least one of the co-runners must present a non-uniform shape. Otherwise, if the bandwidth consumption of both co-runners is uniform, no significant insights will be appreciated on the resultant plot.

Figure 4 presents the results of the described exper-

[2] Notice that the number of reads does not correspond with the number of loads in the x86 ISA. Some instructions (e.g. arithmetic) can access to the cache since the destination or source operand can be a memory location.



(a) *cactusADM* with *h264ref* (b) *bzip2* with *h264ref* (c) *bwaves* with *hmmer*

Fig. 4: IPC, RPC, WPC, and OPC evolution over time when running a pair of benchmarks on a single SMT core.

iment for three pairs of benchmarks for the execution interval ranging from 30 to 130 seconds. For analysis purposes, each plot shows the dynamic evolution of the IPC of a given benchmark, and then differentiates between the RPC, the number of instructions that perform a L1 data cache write per cycle (WPC), and other instructions per cycle (OPC)[3]. Each pair of benchmarks is presented by a figure on the top row of plots and the corresponding one in the bottom row. The pairs of processes that simultaneously run on the same core are *cactusADM* with *h264ref* (Figure 4a), *bzip2* with *h264ref* (Figure 4b), and *bwaves* with *hmmer* (Figure 4c). Note that the benchmarks on the top row present non-uniform L1 bandwidth utilization in stand-alone execution (see Figure 3), while the ones in the bottom row show uniform L1 bandwidth utilization when running without co-runner.

Several observations can be appreciated in this figure that can help design thread allocation policies. First, when a pair of processes runs concurrently on the same core, its L1 bandwidth consumption and IPC significantly drop with respect to that achieved in stand-alone execution. Although such a drop was expected, it is interesting to notice that in some cases this drop is above 40% (e.g., *bwaves* or *cactusADM*, see Figure 3g and Figure 3h, respectively), which shows the importance of the L1 contention point. The second observation is that the IPC and RPC of each process are strongly related with that of its co-runner. In particular, when a thread experiences a drop in the IPC, a positive side effect occurs in the co-runner, which turns into an increase in its number of retired instructions.

A deeper look into the plots reveals more precisely how the co-runners affect each other. For instance, lets focus on the couple *cactusADM* and *h264ref*. The most interesting effect is the one caused by *cactusADM* on the behavior of *h264ref*. The decreasing trend in the IPC of *cactusADM*, in isolated execution, causes a synchronized increasing trend in the IPC of *h264ref* when they run concurrently on the same core. Note that in isolation, *h264ref* shows a uniform IPC. However, the key aspect lies in the RPC, that is, the L1 bandwidth consumption. As the number of committed instructions in *cactusADM* is reduced, so does its RPC, which causes a reduction in the L1 bandwidth consumed by the process. In this way, there is more L1 bandwidth available to *h264ref*, which turns into an increase in its RPC. The IPC improvement is not exclusively caused by the increase in RPC since WPC and OPC also grow. Nonetheless, experimental results show that RPC is usually the component with

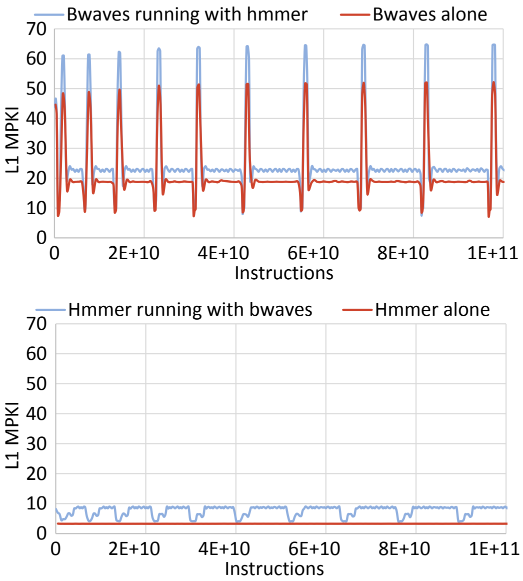


Fig. 5: L1 MPKI evolution over time when running a pair of benchmarks on a single SMT core.

[3] The number of other instructions is calculated as the total number of instructions minus the number of instructions that perform a read or a write in the L1 cache.

highest weight on the overall IPC and presents the most similar shape to the IPC curve among the different studied components.

A similar behavior is observed with the other two pairs of benchmarks. The IPC of *h264ref* when running with *bzip2* grows synchronized with the IPC drop of *bzip2*. Although all the IPC components (RPC, WPC, and OPC) rise, RPC increase is that presenting the greatest magnitude. Similarly, in the last pair of benchmarks, *bwaves* and *hmmer*, the drops of the IPC, and particularly RPC, of *bwaves* leaves more L1 bandwidth available to *hmmer*, which takes advantage of this bandwidth to improve its IPC.

4.3 Impact of Cache Space Contentions on L1 Bandwidth Consumption

The impact of memory resource consumption (bandwidth and space) on shared caches has been addressed in previous work [7], [8], with the aim of estimating the performance of applications when the memory resources are being shared between different processes and thus, their availability is reduced with respect to standalone execution. Previous approaches rely on microbenchmarks, which are synthetic benchmarks that are run concurrently with the target application, but on distinct cores. This way makes performance interferences only to appear on the studied shared resource. Unfortunately, these approaches are not suitable for study space contention on L1 caches in SMT processors, since the microbenchmark and the application should be run on the same core in order to share the same L1 cache; consequently, performance interferences other than L1 cache space will rise.

Unlike previous work, this section tries to provide insights about how L1 cache space contention affects the cache performance of a given benchmark, which turns into a reduction of the L1 bandwidth consumption. For this purpose, we analyze how the L1 misses per kilo instruction (L1 MPKI) of two processes running simultaneously on the same core increases over isolated execution. We use this metric because it is only affected by cache space. That is, neither pipeline resources contention nor cache bandwidth consumption significantly affect the L1 MPKI of a given process. As example, Figure 5 depicts the L1 MPKI corresponding to the co-runners of Figure 4c, both when they run simultaneously on the same core and in standalone execution. Notice that X-axis represents the number of committed instructions instead of time to match, in the figure, the standalone execution of each process with its concurrent execution.

Results show that the L1 MPKI of both processes rise when they run simultaneously due to space contention. As a result of the increase in the L1 MPKI, the out-of-order execution engine cannot hide most of the L1 miss penalty (i.e., latency of extra L2 cache accesses). This fact, jointly with SMT pipeline contention, slowdowns the execution time. Therefore, IPC and RPC, that is, L1 bandwidth consumption, decrease.

This conclusion can be confirmed by the fact that L1 MPKI rises and drops in Figure 5 are synchronized with reductions and increases, respectively, of the L1 bandwidth consumption in Figure 4c. In summary, bandwidth variation takes into account both L1 bandwidth and cache space contention; therefore, bandwidth utilization can serve as a good indicator of performance degradation due to L1 cache contention.

5 Bandwidth-Aware On-Line Scheduler

With multiprogrammed workloads and different levels of resource sharing, task scheduling is usually carried out in two main steps. In the first step, called process selection, the set of processes to be executed in the next quantum is selected. In the second step, called process allocation, each selected process is mapped to a hardware thread of the processor. In a multithreaded CMP, all the processes selected to be run the next quantum will share the main memory bandwidth, but only the subset of processes assigned to a given core will share its L1 bandwidth. Thus, each scheduling step is responsible for a resource sharing level.

Algorithm 1 presents the main steps of the Bandwidth-Aware On-Line Scheduler (BAOS) proposed. To address bandwidth contention at the two discussed contention points, the proposal consists of a process selection and allocation that are aware of the main memory and L1 bandwidth requirements, respectively. That is, the devised policies guide the scheduling decisions based on the predicted bandwidth utilization of the processes at their corresponding level of the memory hierarchy.

With the purpose of providing the estimations of these bandwidth utilizations, a third step is included. This step makes use of performance counters to collect, for each individual process that was run during the last quantum, its number of L1 and main memory accesses, as well as its number of executed cycles. The collected values are used to calculate the transaction rates per microsecond for the main memory bandwidth (TR$_{MM}$) and L1 bandwidth (TR$_{L1}$) performed by each process.

The bandwidth utilization of a given process during the last quantum is used as the predicted bandwidth utilization for its next execution quantum. Such a simple prediction has shown adequate accuracy. For example, the L1 bandwidth utilization during a given quantum differs, on average for all the SPEC CPU2006 benchmarks, about 5.5% from the utilization in the previous one.

Algorithm 1 BAOS scheduler: main steps

```
Process selection - Aware of MM bandwidth requirements
Process allocation - Aware of L1 bandwidth requirements
Update the bandwidth requirements for the next quantum
of each process $p$ executed in the previous quantum:
    - Gather consumed L1 bandwidth (TR$^p_{L1}$)
    - Gather consumed main memory bandwidth (TR$^p_{MM}$)
```

5.1 Main Memory Bandwidth-Aware On-Line Process Selection

As discussed in Section 4.1, when running multiprogrammed workloads with significant memory requirements, main memory bandwidth contention inflicts important performance degradation. Such a degradation can even exceed 50% of the IPC of the processes, which illustrates the magnitude of the problem. Therefore, it is interesting to design a process selection policy aware of the main memory bandwidth requirements of the processes to mitigate these performance drops.

The main goal of the devised process selection policy consists in evenly distributing the amount of main memory accesses that all the processes of the workload perform throughout its complete execution. By balancing the memory transactions along the execution time, the policy tries to minimize the contention in the main memory access, and prevents most of the memory transactions to be performed in a subset of the quanta suffering high contention, while the memory is much less stressed in other quanta. The proposed policy shares the key idea of distributing the memory accesses along the execution time with the scheduler proposed by Xu et al. [3]. Nevertheless, while Xu's proposal requires prior knowledge of the main memory bandwidth requirements of the processes before running them with the scheduler, the policy we devise works without requiring any prior information.

To balance the main memory transactions during the execution time, the policy makes use of the On-line Average Transaction Rate (OATR), which defines the overall main memory requests that should be performed at the next quantum in order to evenly distribute them along the execution time. The IABW calculated by Xu et al. to distribute the memory requests over the workload execution time [3] is fixed before mix execution since it is calculated with prior information about main memory requirements and execution time of the workload. On the contrary, the OATR changes dynamically during the workload execution based on the changes in the average main memory bandwidth utilization of the processes, which is calculated after each quantum expires.

As the execution progresses, the OATR reaches a value that is more realistic than the IABW since it is calculated using the bandwidth utilization gathered while the processes run concurrently. This is unlike the IABW, which is calculated off-line from the bandwidth utilization measured in stand-alone execution. Xu et al. correct this issue using a polynomial regression, which is not required by the OATR.

Algorithm 2 On-Line Process Selection Policy

```
1: Update the average main memory bandwidth utilization (BW$^p_{MM}$) for each process $p$ of the workload
2: Calculate the OATR:
   $OATR = \frac{\sum_{p=0}^{N} BW^p_{MM}}{N} * \#\text{CPUs}$
3: Select the process $p$ at the process queue head and set BW$_{remain} = OATR - TR^p_{MM}$, CPU$_{remain}$ = $\#CPUs - 1$
4: while # selected process < #CPUs do
5:     Select the process $p$ that maximizes:
       FITNESS(p) = $\frac{1}{\left|\frac{BW_{Remain}}{CPU_{Remain}} - TR^p_{MM}\right|}$
6:     Update BW$_{Remain}$ and CPU$_{Remain}$
7: end while
```

The pseudocode of the process selection policy is presented in Algorithm 2. The first step updates the BW$_{MM}$ of each process of the workload with its average TR$_{MM}$ of the previously executed quanta. Next, the second step calculates the OATR as the average BW$_{MM}$ of the processes of the workload, multiplied by the number of hardware threads of the experimental platform. The resultant OATR is used in the algorithm as the target bandwidth utilization that the processes running in the next quantum should achieve to balance the main memory accesses along the workload execution time.

The first process selected to run in the next quantum is chosen in the third step. To avoid process starvation, mainly due to extreme bandwidth requirements of a given benchmark, the process at the head of the process queue, which is the one not executed for longer, is always selected to run. After that, both the remaining bandwidth (BW$_{Remain}$) and unallocated hardware threads (CPU$_{Remain}$) are updated accordingly. The remaining processes are selected in the next steps using the fitness function (fifth step) until all the hardware threads are allocated or no more processes are in the process queue.

The fitness function quantifies, for each process p, the gap between its predicted main memory transaction rate for the next quantum (TR$^p_{MM}$) and the average bandwidth remaining for each unallocated hardware thread (BW$_{Remain}$/CPU$_{Remain}$). The process with the best fit is the one that maximizes the fitness function and it is selected to run during the following quantum, updating the BW$_{Remain}$ and CPU$_{Remain}$ variables accordingly.

Due to the lack of previous information, the scheduler has to face a cold start the first quanta of the execution of a new workload, since it has no prior information about the processes. Besides, the average main memory transaction rate of the processes (BW$_{MM}$) can take a few quanta to reach a dependable value, which can increase the length of such cold start. To mitigate a possible negative impact on performance, we propose to let the Linux kernel drive the scheduling decisions during a few quanta at the beginning of the execution, while the proposed scheduler collects enough bandwidth utilization information of the processes. We found experimentally that a short period of about thirty quanta (over executions that last more than five thousand quanta) is large enough to avoid significant performance losses.

5.2 Dynamic L1 Bandwidth-Aware Process Allocation

The analysis presented in Section 4.2 illustrates that the high L1 bandwidth utilization of two threads running on the same core may produce significant performance degradation. Thus, the allocation of the processes to be run simultaneously on a given core has an important influence on their performance. Additionally, it should also be considered the fact that their bandwidth requirements can widely vary over the execution time. To address these issues, this section proposes a process allocation policy that is aware of the L1 bandwidth requirements of the processes. Notice that, although overall SMT contention is addressed with the proposed process allocation, it uses the L1 bandwidth utilization to determine the allocation of processes to cores.

The key idea of the process allocation policy consists in balancing the overall L1 bandwidth utilization of the running processes (they have been previously selected to run by the process selection policy) among all the processor cores. In this way, the policy tries to promote thread to core mappings that do not saturate the available L1 bandwidth of some cores while it is underused in others.

The L1 bandwidth aware process allocation used in this scheduler is based on the Dynamic t2c policy we proposed in [22]. In [22] the policy itself was responsible for gathering the L1 bandwidth consumption of the processes, but in the current implementation this part has been moved to Algorithm 1.

A dynamic process allocation policy presents two main advantages with respect to a static process allocation policy (also described in [22]). A dynamic policy is more convenient than a static one since it does not require prior information on the processes. Furthermore, it also reacts to non-uniform shapes in the consumed L1 bandwidth. For instance, L1 bandwidth requirements of benchmarks like *astar* or *mcf* can be properly addressed. That is, the dynamic policy can allocate to the same core *astar*, when it presents low L1 bandwidth requirements, together with a process with high L1 bandwidth consumption. And then, when *astar* increases its bandwidth utilization, the policy changes its co-runner to run *astar* with a process with lower bandwidth requirements.

Algorithm 3 presents the pseudocode of the proposed process allocation policy. Since the experimental platform supports simultaneous execution of only two threads in each core, finding the thread to core assignment that achieves the optimal balance of L1 bandwidth consumption among cores is simplified. For instance, threads can be ordered according to their TR_{L1} (first step). The RPC used to study the effects of L1 bandwidth contention on performance could be used in this algorithm since it is actually the same metric expressed in different units. However, the algorithm uses TR_{L1} for consistency reasons. Then, the threads with highest and lowest L1 bandwidth requirements are assigned to the same core (third and fourth steps). This rule is iteratively applied to obtain the remaining pairs of co-runners. Notice that the maximum number of threads that must be sorted each time the policy is executed is equal to the number of hardware threads, since only the processes selected by the process selection policy are considered. This restriction limits the computational cost of sorting the processes, which has been measured experimentally and is negligible compared with the quantum length and the benefits provided by a good thread to core assignment.

Algorithm 3 Dynamic Process Allocation Policy

```
1: Sort the selected processes in ascending $\text{TR}_{L1}$
2: **while** there are unallocated processes **do**
3:     Select the processes $\text{P}_{head}$ and $\text{P}_{tail}$ with maximum and
       minimum bandwidth requirements
4:     Assign $\text{P}_{head}$ and $\text{P}_{tail}$ to the same core
5: **end while**
```

If the SMT processor supports the execution of three or more threads it is possible to balance L1 requirements following a similar approach to that one explained for the process selection policy. The alternative algorithm would calculate the cumulative TR_{L1} of all the threads that have been selected to run the next quantum and would divide this value by the number of cores. Then, threads could be properly allocated to the cores in order to balance the TR_{L1} differences among L1 caches using a fitness function.

Finally, remark that the number of process migrations among cores is not limited by the proposed policy. Although, an overhead is incurred when migrating the architectural state of the process and extra time is wasted warming up the L1 cache, we found that such overhead is negligible when working with long quanta like the ones used by modern operating systems [23].

6 Evaluation Methodology

To evaluate the performance of the BAOS scheduler, both process selection and process allocation policies have been implemented in a user-level scheduler that controls which processes are allowed to run and sets their core affinities. For performance comparison purposes, the Linux process selection and process allocation policies, as well as one state-of-the-art process selection policy, explained in Section 7.1, and two state-of-the-art process allocation policies, explained in Section 7.2, have also been considered.

By implementing a single user-level scheduler framework shared by all the policies, we ensure that any possible overhead incurred by process management or handling performance counters is the same for the studied schedulers, so offering a fair comparison.

The Linux policies correspond to the Completely Fair Scheduler (CFS) [24]. In summary, this scheduler tries to give the same CPU utilization to all the processes, keeping process affinity to cores as much as possible in order to avoid constant process migrations. To set the Linux process selection policy, all the processes are allowed to run, which lets the Linux kernel to decide which processes effectively run during each quantum. Similarly, to study the Linux process allocation policy, the affinities of the processes have been configured to allow any process to run on any core, so the Linux kernel finally determines the process allocation.

As mentioned in Section 3, the execution time of the benchmarks when building the mixes is fixed to 200 seconds in stand-alone execution [3]. The number of instructions that each benchmark runs during this time is measured offline, and the scheduler finishes a given benchmark execution when it surpasses its target number of instructions. In this way, we avoid the benchmarks to present different weights in the mix execution. Otherwise, the performance of a given mix would be better when running more instructions from benchmarks with higher IPC. Furthermore, fixing the benchmark execution time also prevents that scheduling policies prioritizing long jobs first could bring better performance [3]. Finally, we try to minimize the number of quanta where there are less runnable processes than hardware threads, which would reduce contention.

A set of twelve mixes has been designed to evaluate the scheduler performance. Each mix consists of twenty four benchmarks, that is, the number of processes doubles the available hardware contexts. See Section 1 of the supplementary material for further details of the mix composition.

6.1 Performance Metrics

A wide set of metrics has been analyzed for evaluation purposes. First we use the average IPC of the threads composing a workload. This is the plain metric to compare throughputs. Unfair scheduling strategies may favor this metric if they prioritize the execution of those benchmarks with highest IPC [25]. These scenarios, however, are not allowed on the described evaluation methodology. To deal with fairness, the harmonic mean of weighted IPC [26] is also used, which captures fairness additionally to performance.

In addition to these metrics the turnaround time of the mix execution has also been evaluated. This metric is widely used because it refers to the elapsed time since the mix is launched until the last process finishes. Unfortunately, the turnaround time does not take into account the fact that at the end of the mix execution the number of running processes will probably be lower than the number of hardware threads of the processor. As illustrated in Figure 1, these free hardware threads could be used to run other workloads. To consider them in the evaluation, we define the *consumed slots* metric as the accumulated number of hardware threads used in each OS quantum required to complete the execution of a given workload. Notice that the consumed slots is a more meaningful metric than the turnaround time, since it gives lower weight to the quanta where the number of running processes is lower than the number of hardware threads.

7 Performance Evaluation

First, we analyze the performance benefits provided by both the proposed process selection and the process allocation policies in an isolated way. Then, we study the performance of the complete scheduler with respect to Linux.

The different performance evaluation studies can be carried out by properly selecting the desired policies in the implemented scheduling framework shared by all the policies. This way allows a fair comparison since the policy to be analyzed is the only difference between the two studied scheduling approaches. When evaluating the policies in isolation, the scheduling step not being analyzed is set to the Linux policy.

The plotted results in all the experiments correspond to the average values of twenty executions and 95% confidence intervals.

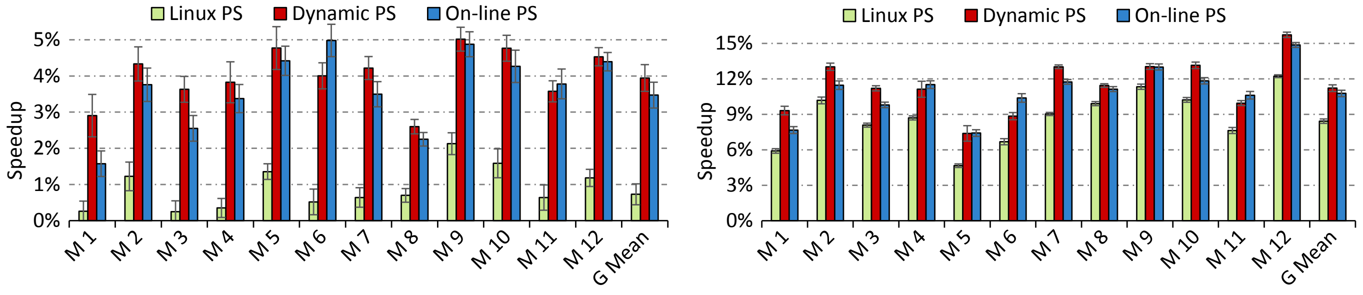
7.1 Process Selection Policies Evaluation

In this section, the performance of the designed On-line process selection policy is compared to that achieved by a dynamic process selection policy based on Xu's scheduler [3] and the Linux policy implemented in the Completely Fair Scheduler [24], relative to the performance of the *naive* process selection policy.

The Dynamic policy distributes the amount of main memory accesses that all the processes of the running mix perform over the execution time of the mix. However, unlike the devised On-line process selection policy, the Dynamic policy requires to know the main memory transaction rate and execution time of all the processes of the mix prior to its execution. This information is used to calculate the IABW, which represents the target bandwidth that the selected processes to run at a given quantum should achieve to balance the main memory accesses along the mix execution time. Unfortunately, the use of prior information of the processes implies an important drawback, since it makes the approach impractical on real systems.

The naive process selection policy is implemented as a random policy, which yields the main memory bandwidth consumption to vary over a broad range during each quantum. Thus, this policy presents executions combining periods with high bandwidth contention and periods where the available bandwidth is underutilized.

Figure 6 presents the speedups achieved by the Linux, Dynamic and On-line process selection policies relative to the naive policy regarding IPC-based metrics. Results regarding the average IPC metric (Figure 6a) show that the Dynamic, and On-line policies improve the performance of the Linux and naive policies. The speedups achieved by the Dynamic and On-line policies usually fall in between 3% and 5%, being higher for the Dynamic policy in all the mixes but two. With regard to the Linux



(a) Average IPC

(b) Harmonic mean of weighted IPC

Fig. 6: Speedup of the process selection (PS) policies with respect to the naive policy.

policy, it achieves much lower speedups since only mix 8 exceeds 2%.

Figure 6b depicts the speedups of the policies regarding the harmonic mean of weighted IPC, which in addition to performance evaluates fairness. The achieved speedups with this metric are much higher for the three evaluated policies with respect to the naive policy, which indicates that the Linux, Dynamic and On-line policies perform a much fairer process selection. The Dynamic policy achieves the best performance, showing the highest speedup in eight mixes and an average speedup of 11.4% across the evaluated mixes. Close to its performance, the On-line policy achieves the best speedup in four mixes, with an average speedup about 11%. Linux achieves the worst speedup relative to the naive policy with an average value of 8.7%.

Finally, Figure 7 presents the speedups regarding the turnaround time of the mixes. Results show that all the process selection policies widely improve the performance of the naive policy with speedups that usually exceed 12%. The reduction in the time required to complete the execution of the mixes shows the significance of the main memory bandwidth contention point and how smart policies can mitigate such contention and improve the performance. Comparing the performance of the evaluated policies, results suggest that Linux performs worse than the Dynamic and On-line process selection policies, since it achieves significantly lower speedup in mixes like 2, 3, 6 or 10. Regarding the Dynamic and On-line policies, we can see that the On-line policy achieves better performance than the Dynamic policy in eight mixes. In addition, the average speedup for the evaluated mixes is 12.6% and 12.8% for the Dynamic and On-line policies, respectively, which shows that the On-line policy performs slightly better.

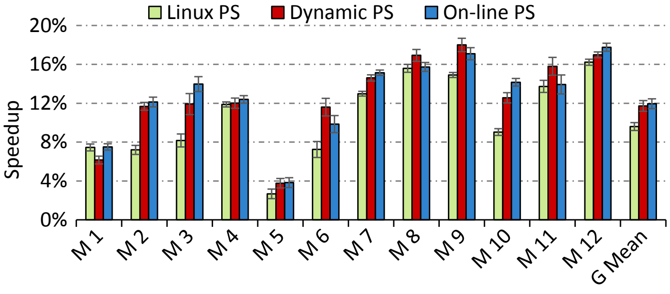


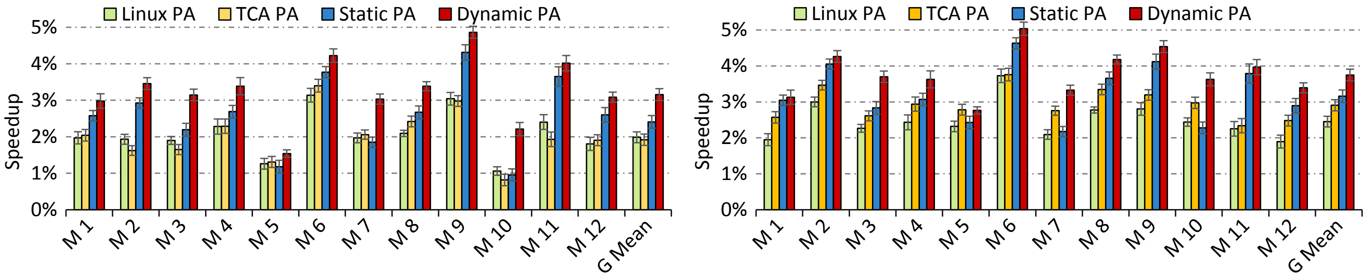
Fig. 7: Speedup of the process selection policies (PS) w.r.t. the naive policy regarding turnaround time.

The achieved speedups regarding the turnaround time help explain the relatively low speedups observed with the average IPC metric. Notice that the process selection of the naive policy significantly enlarges the execution time of the mixes, which causes the distribution of the overall main memory accesses in a longer interval, so reducing the contention. In this way, the processes see its performance improved and the average IPC of the mix is enhanced, but it is not a desirable behavior since it is achieved at the expense of a higher turnaround time.

In summary, the three process selection policies evaluated significantly improve the performance of the naive policy, with speedups that usually exceed 10% regarding the harmonic mean of weighted IPC and turnaround time metrics. Among the policies, the best results are obtained with the Dynamic and On-line policies that perform better than Linux in all the evaluated mixes. Finally, notice that the On-line policy is able to achieve performance comparable to (if not better than) that achieved by the Dynamic policy, despite the fact this policy uses bandwidth information obtained in prior executions of the processes to calculate the IABW. This can be explained by the fact that the bandwidth information used by the Dynamic policy is gathered in stand-alone execution, and thus despite being representative of the bandwidth requirements of the processes, it loses some accuracy when running with co-runners, because it does not consider the interferences that affect their bandwidth utilization.

7.2 Process Allocation Policies Evaluation

In this section, the performance of the Dynamic thread allocation policy used in the proposed scheduler is compared against that of the Static policy [22], a state-of-the-art policy proposed by Acosta et al. [15] (from now on referred to as TCA policy), and the Linux thread allocation policy of the Completely Fair Scheduler [24], with respect to the performance of a naive process allocation policy.



(a) Average IPC

(b) Harmonic mean of weighted IPC

Fig. 8: Speedup of the process allocation (PA) policies with respect to the naive policy.

Regarding the TCA policy, Acosta et al. [15] presented two thread allocation policies designed for robust and naive IFetch policies, respectively. The authors stated that the main reason that causes a negative interaction between two threads running concurrently on a core is their memory behavior and ILP, thus, they used the IPC of each process, as an estimation of both of them, to guide the thread to core mapping. To implement the algorithm, the IPC of each process during its last execution quantum, measured with performance counters, is used as predicted IPC for the next quantum. It replaces the IPC prediction mechanism devised by the authors, which works similarly but does not consider the interferences that the co-runners can cause to the IPC of a given process.

The Static policy allocates threads to cores in the same way as the Dynamic policy, but using the average L1 bandwidth consumption of the processes measured in stand-alone execution. Thus, the policy must know this information for each process, which reduces its suitability for real situations. Finally, as done in the evaluation of the process selection policies, the naive process allocation policy is implemented as random. Therefore, during a single quantum, one core can present high L1 bandwidth utilization causing important contention while other core's L1 bandwidth is underutilized.

Figure 8 compares the performance of the different process allocation policies with respect to the naive policy using IPC-based metrics. Figure 8a presents the achieved speedups of the average IPC, while Figure 8b shows the speedups of the harmonic mean of weighted IPC.

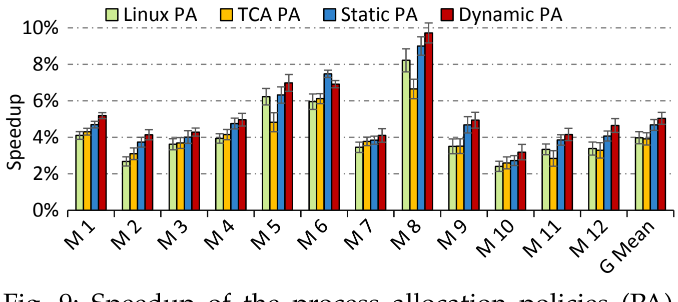


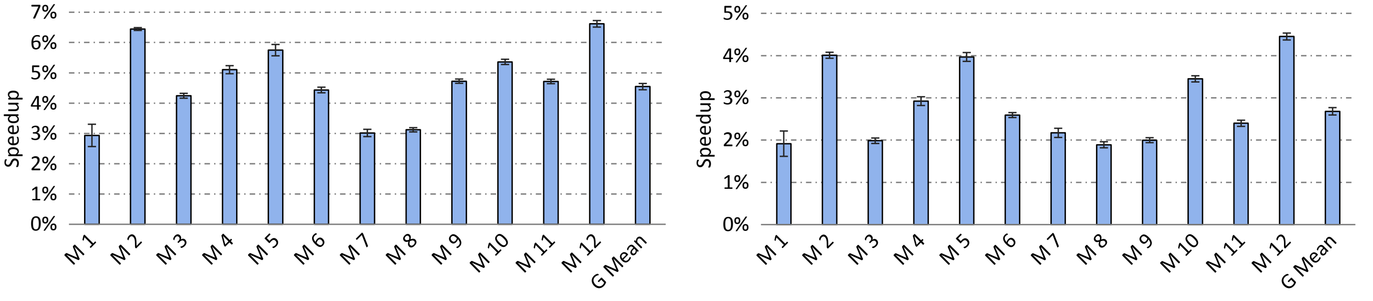
Fig. 9: Speedup of the process allocation policies (PA) w.r.t. a naive policy regarding turnaround time.

Figure 8a shows that the Dynamic process allocation policy achieves the best performance in all the mixes. Speedups relative to the naive process allocation fall in between 3% and 5% (except for mixes 5 and 10), while the Static policy only achieves speedups above 3% in three mixes. This fact shows that the Static policy offers good performance in some mixes where there is higher uniformity in the L1 bandwidth requirements of the processes, but its results are much worse in other mixes where a higher number of processes present non-uniform shapes. Finally, the Linux and TCA thread allocation policies get similar performance but always lower than the Dynamic policy.

The differences among the policies are reduced when considering fairness, as shown in Figure 8b. The Dynamic process allocation policy achieves speedups above 2% in all mixes except three of them, while the Static, TCA, and Linux policies fall below 2% in seven, nine and eight mixes, respectively. Using this metric, we can see that the Linux policy performs better than the TCA policy, which means that the TCA policy does not equally distribute the performance degradation among all the processes, but unfairly damages the performance of some of them above the others.

Figure 9 compares the achieved speedup regarding the turnaround time. At a first glance, the results show that the speedups are increased compared to those achieved with the IPC-based metrics. The plot confirms that the Dynamic policy achieves the highest performance among the process allocation policies evaluated, with a maximum speedup close to 10% and an average value of 5.3%, while the Static policy achieves an average speedup of 4.9%. The Linux and TCA policies achieve a lower speedup with average value about 4.1%.

In short, the Dynamic process allocation policy achieves the highest performance in both IPC-based and time-based metrics. Since this policy does not require prior information of the processes but collects the L1 bandwidth utilization of the processes at runtime using performance counters, we can conclude that this is the most adequate process allocation policy for the designed scheduler.



(a) Average IPC

(b) Harmonic mean of weighted IPC

Fig. 10: Speedup of the proposed scheduler relative to Linux scheduler with IPC based metrics.

7.3 BAOS Scheduler Evaluation

This section analyzes the performance of the BAOS scheduler proposed (On-line process selection and Dynamic process allocation) with respect to the Linux scheduler.

Figure 10 presents the performance benefits reached using the IPC-based metrics described above. Figure 10a shows the speedup of the average IPC achieved by the BAOS scheduler over the Linux scheduler for the studied mixes. The proposed scheduler improves Linux in all mixes, with speedups ranging from above 3% to above 7%, and with nine of twelve mixes achieving over 4% speedup and five exceeding 5%. Since the average IPC is a metric focused on performance, the results show that the proposed scheduler effectively addresses bandwidth contention at the L1 cache and main memory, which results in a significant performance increase.

To ensure that performance improvements are not unfairly obtained by favoring the execution of certain processes, Figure 10b compares the speedups with the harmonic mean of weighted IPC metric. The BAOS scheduler achieves speedups ranging from around 2% to 4.5% with respect to the Linux scheduler. Although they are slightly reduced compared to those obtained with the average IPC metric, they show that the proposed scheduler works fairer than the Linux scheduler in addition to improve its performance.

Figure 11 presents the speedup achieved by the BAOS scheduler regarding the turnaround time. The plot shows that the proposed scheduler shortens the execution time of all the evaluated mixes with speedups over 2% (except mix 9). Five mixes achieve a speedup between 3% and 4%.

Notice that dealing with bandwidth contention, the improvements achieved in throughput, as the average IPC speedups, do not directly correspond to reductions in the turnaround time of the mixes. In fact, when the turnaround time of the mix is shortened the bandwidth contention rises, since the same number of memory or cache accesses are concentrated in a shorter period of time. In contrast, sometimes the throughout is enhanced at the expenses of a longer execution time since the memory requests have more time to be distributed. An important advantage of the proposed scheduler is that it improves the throughput without enlarging the turnaround time of the mixes.

The growth of the confidence intervals in the turnaround time speedups is caused by the high variability of the turnaround time of the mixes with the Linux scheduler. For instance, the typical deviation of the turnaround time of different executions of mix 2 with the Linux scheduler triples the one obtained by the BAOS scheduler.

Next, the analysis focuses on how the BAOS and Linux schedulers consume the execution slots dynamically and how the slots are being released at the end of the execution, where the number of remaining processes is lower than the number of hardware threads. We define an execution slot as an available hardware threads. Thus, a consumed slot means that one thread is running on a given slot.

Figure 12 presents the evolution of the consumed slots during the execution of a subset of the studied mixes, which shows how the hardware contexts are released earlier with the BAOS scheduler (due to space constraints the plots for the remaining mixes are presented in Section 2 of the supplementary material). The plot for each mix presents in the y-axis the number of consumed slots, that is, the number of threads running at each OS quantum. It ranges from twelve, the maximum number of threads that can run simultaneously in the experimental platform (six dual-thread cores), to zero, which is the point where there are no more processes pending (i.e., the workload execution finishes). The x-axis represents the sequence of OS quanta, which allows

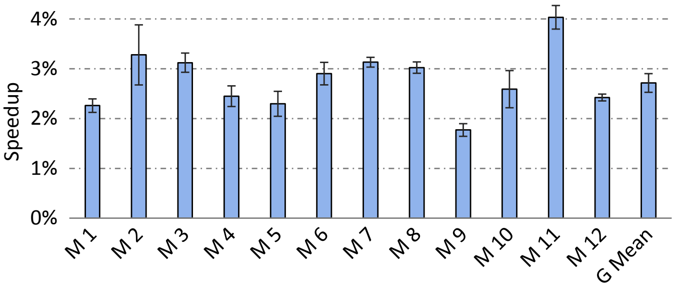
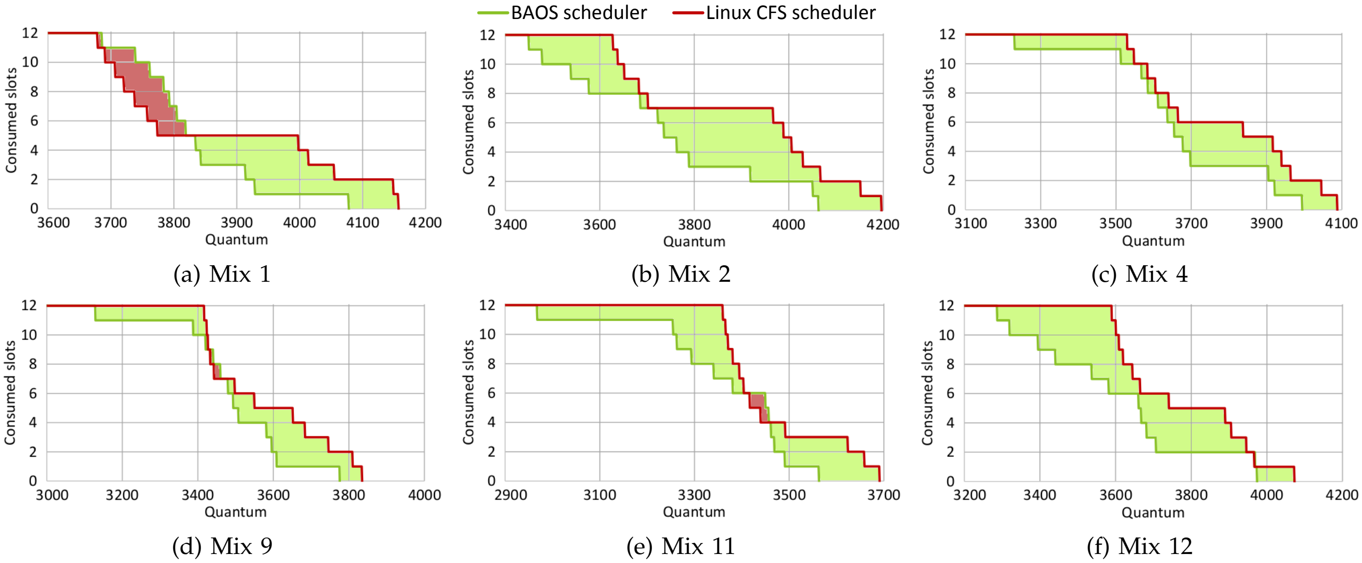


Fig. 11: Speedup of the proposed scheduler w.r.t. Linux scheduler using the turnaround time metric.



(a) Mix 1

(b) Mix 2

(c) Mix 4

(d) Mix 9

(e) Mix 11

(f) Mix 12

Fig. 12: Consumed slots in the workloads. The proposed scheduler saves slots in the green (light) area, while Linux does it in the red (dark) area.

comparing the measured execution time.

The plots show that the benefits provided by the BAOS scheduler, area colored in green (or light gray), go beyond the reduction in the turnaround time of the mixes. The proposed scheduler usually finishes the processes that form a workload earlier, allowing the scheduler to put some cores into a low power state or use them to run a different workload. Notice that an early completion of the processes can only be achieved without enlarging the execution time of the mixes by reducing the bandwidth contention along the memory hierarchy, which is the main goal of the proposed scheduler.

For instance, Figure 12b, Figure 12e and Figure 12f present the evolution of the consumed slots of mixes 2, 11 and 12, which showed the highest speedups with the previous metrics. As observed, the BAOS scheduler significantly reduces the number of slots required to complete the execution of the mixes.

On the other hand, Figure 12a, Figure 12c and Figure 12d present the consumed slots plots for some mixes that showed the lowest speedups in the metrics previously studied. Even in these cases, the BAOS scheduler is able to bring forward the completion of the processes with respect to the Linux scheduler, saving a noticeable amount of execution slots. Note that in mix 1 (Figure 12a), although Linux saves more execution slots from quantum 3700 to 3800 approximately (bounded by the area shaded in red color (dark grey)), the proposed scheduler saves a higher number of slots through the overall execution, which compensates this loss. With a lower magnitude, the same effect can also be observed in mixes 9 and 11.

8 Conclusions

In this work, we have addressed the bandwidth contention at main memory and L1 caches in commercial multithreaded multicore processors.

Regarding main memory contention, the experiments carried out in this work have shown its significance in the experimental platform, where performance degradation can drop by 50% the IPC of some processes. On the other hand, regarding L1 bandwidth contention, two interesting findings have been observed in the designed experiments: i) performance and L1 bandwidth consumption of a given process follow the same shape over the execution time regardless the process runs in stand-alone execution or with co-runners, and ii) when two processes run simultaneously on a multithreaded core, the implicit drops in the L1 bandwidth and IPC of a process trigger the opposite effect in the co-runner.

To deal with the observed bandwidth contention, a scheduling algorithm has been proposed aimed at reducing bandwidth contention. The devised On-line scheduler consists of two main policies: process selection and process allocation. First, the former policy addresses the main memory bandwidth contention distributing the main memory accesses of the processes of a workload along its execution time. Then, the latter policy tackles L1 bandwidth contention by balancing the L1 accesses of the selected processes among the L1 caches. Both policies *only* use the bandwidth consumption of the processes, gathered with performance counters, to guide the scheduling. Thus, no prior information of the running processes is required, which represents an important feature of this scheduler that makes it feasible for real systems.

Experimental evaluation on a Xeon E5645 has shown that the proposed scheduler mitigates bandwidth contention at both L1 cache and main memory. Compared to the Linux scheduler, performance benefits (i.e., IPC) rise up to 6.7%, with a geometric mean of speedups by 4.6%. In addition, the devised scheduler works fairer than Linux, with speedups of the harmonic mean of weighted IPC ranging from 1.9% to 4.4%.

Acknowledgments

This work was supported by the Spanish Ministerio de Economía y Competitividad (MINECO) and by FEDER 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] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou, "Realistic Workload Scheduling Policies for Taming the Memory Bandwidth Bottleneck of SMPs," in *HiPC*, 2004, pp. 286–296.

[3] D. Xu, C. Wu, and P.-C. Yew, "On Mitigating Memory Bandwidth Contention Through Bandwidth-Aware Scheduling," in *PACT*, 2010, pp. 237–248.

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

[5] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Shared Resource Contention in Multicore Processors Via Scheduling," in *ASPLOS*, 2010, pp. 129–142.

[6] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "Understanding Cache Hierarchy Contention in CMPs to Improve Job Scheduling," in *IPDPS*, 2012, pp. 508 –519.

[7] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten, "Cache Pirating: Measuring the Curse of the Shared Cache," in *ICPP*, 2011, pp. 165–175.

[8] M. Casas and G. Bronevetsky, "Active Measurement of Memory Resource Consumption," in *IPDPS*, 2014, pp. 995–1004.

[9] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using OS Observations to Improve Performance in Multicore Systems," *IEEE Micro*, vol. 28, no. 3, pp. 54–66, may 2008.

[10] A. Fedorova, M. Seltzer, and M. D. Smith, "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler," in *PACT*, 2007, pp. 25–38.

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

[12] A. Settle, J. Kihm, A. Janiszewski, and D. Connors, "Architectural Support for Enhanced SMT Job Scheduling," in *PACT*, 2004, pp. 63–73.

[13] S. Eyerman and L. Eeckhout, "Probabilistic Job Symbiosis Modeling for SMT Processor Scheduling," in *ASPLOS*, 2010, pp. 91–102.

[14] V. Čakarević, P. Radojković, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Characterizing the Resource-Sharing Levels in the UltraSPARC T2 Processor," in *MICRO*, 2009, pp. 481–492.

[15] C. Acosta, F. J. Cazorla, A. Ramirez, and M. Valero, "Thread to Core Assignment in SMT On-Chip Multiprocessors," in *SBAC-PAD*, 2009, pp. 67–74.

[16] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero, in *Transactions on High-Performance Embedded Architectures and Compilers III*, 2011, ch. Dynamic Cache Partitioning Based on the MLP of Cache Misses, pp. 3–23.

[17] S. Srikantaiah, M. Kandemir, and Q. Wang, "SHARP Control: Controlled Shared Cache Management in Chip Multiprocessors," in *MICRO*, 2009, pp. 517–528.

[18] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic Shared Cache Management (PriSM)," in *ISCA*, 2012, pp. 428–439.

[19] K. J. Nesbit, J. Laudon, and J. E. Smith, "Virtual Private Caches," in *ISCA*, 2007, pp. 57–68.

[20] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moreto, D. Chou, et. al., "Tessellation: Refactoring the OS Around Explicit Resource Containers with Continuous Adaptation," in *Design Automation Conference (DAC)*, 2013, pp. 1–10.

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

[22] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "L1-Bandwidth Aware Thread Allocation in Multicore SMT Processors," in *PACT*, 2013, pp. 123–132.

[23] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. Wenisch, et. al., "Composite Cores: Pushing Heterogeneity Into a Core," in *MICRO*, 2012, pp. 317–328.

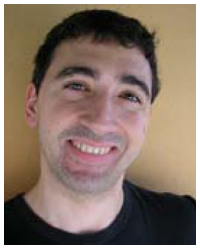
[24] I. Molnar, "Modular Scheduler Core and Completely Fair Scheduler [CFS]," *http://lwn.net/Articles/230501/*.

[25] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor," in *ASPLOS*, 2000, pp. 234–244.

[26] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Thoughput and Fairness in SMT Processors," in *ISPASS*, 2001, pp. 164–171.

Josué Feliu received the BS and MS degrees in computer engineering from the Universitat Politècnica de València (UPV), Spain, in 2011 and 2012, respectively. He is currently working towards a PhD degree at the Department of Computer Engineering (DISCA) of the same university. His PhD research focuses on scheduling strategies for multicore, multithreaded and future heterogeneous manycore processors.

Julio Sahuquillo received his BS, MS, and PhD degrees in Computer Engineering from the UPV, Spain. Since 2002 he is an associate professor at the DISCA department at the UPV. He has published more than 100 refereed conference and journal papers. His current research topics include multi- and manycore processors, memory hierarchy design, and power dissipation. He has cochaired several workshops, collocated in conjunction with IEEE supported conferences.



Salvador Petit received the PhD degree in computer engineering from the UPV, Spain. Currently, he is an associate professor in the DISCA department at the UPV where he has taught several courses on computer organization. His research topics include multithreaded and multicore processors, memory hierarchy design, as well as real-time systems.

José Duato received the MS and PhD degrees in electrical engineering from the UPV, Spain. He is currently a professor with the DISCA department at the UPV. He has published more than 380 refereed papers. He proposed a powerful theory of deadlock-free adaptive routing for wormhole networks. Versions of this theory have been used in the design of the routing algorithms for the MIT Reliable Router, the Cray T3E supercomputer, the internal router of the Alpha 21364 microprocessor, and the IBM BlueGene/L supercomputer.