Feliu Pérez, J.; Petit Martí, SV.; Sahuquillo Borrás, J.; Duato Marín, JF. (2014). Cache-Hierarchy contention-aware scheduling in CMPs. IEEE Transactions on Parallel and Distributed Systems. 25(3):581-590. doi:10.1109/TPDS.2013.61.

Cache-hierarchy contention aware scheduling in CMPs

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

Abstract—In order to improve CMP performance, recent research has focused on scheduling strategies to mitigate main memory bandwidth contention. Nowadays, commercial CMPs implement multi-level cache hierarchies that are shared by several multithreaded cores. In this microprocessor design, contention points may appear along the whole memory hierarchy. Moreover, this problem is expected to aggravate in future technologies, since the number of cores and hardware threads, and consequently the size of the shared caches increases with each microprocessor generation.

This paper characterizes the impact on performance of the different contention points that appear along the memory subsystem. The analysis shows that some benchmarks are more sensitive to contention in higher levels of the memory hierarchy (e.g., shared L2) than to main memory contention.

In this paper we propose two generic scheduling strategies for CMPs. The first strategy takes into account the available bandwidth at each level of the cache hierarchy. The strategy selects the processes to be co-scheduled and allocates them to cores in order to minimize contention effects. The second strategy also considers the performance degradation each process suffers due to contention aware scheduling. Both proposals have been implemented and evaluated in a commercial single-threaded quad-core processor with a relatively small two-level cache hierarchy. The proposals reach, on average, a performance improvement by 5.38% and 6.64% when compared with the Linux scheduler while this improvement is by 3.61% for an state-of-the-art memory-contention aware scheduler under the evaluated mixes.

Keywords—contention-aware scheduling; contention-points; shared caches; cache hierarchy; memory-contention;

1 Introduction

Multi-core processors have become the common implementation for high-performance microprocessors. These Chip MultiProcessors (CMP) incorporate additional cores on the same chip with each technology generation, and they have the potential to provide higher levels of processing performance than their single-core counterparts, while attacking power, cooling and package costs problems.

Most of these CMPs are Symmetric MultiProcessing (SMP) systems, whose main performance bottleneck lies in the interconnection between the computational multi-core chip and the main memory. In most processors, the most important component of this bottleneck has typically been the main memory latency. However, as the number of cores and their multithreading capabilities increase, the contention for the available main memory bandwidth is becoming a major concern since it prevents current and future many-core designs from scalability.

When the number of jobs exceeds the number of cores, bandwidth contention aware strategies can help the scheduler to reduce main memory contention by avoiding the concurrent execution of memory-hungry applications. From now on, we will use the term bandwidth contention or simply contention to refer to the contention caused by bandwidth constraints. The strategies take into account the total bandwidth required by applications and schedule a set of them to execute concurrently, considering that their accumulated bandwidth requirements do not exceed the available bandwidth. Otherwise, performance could severely be damaged due to main memory contention. Nevertheless, previous research has shown that the scheduler must try to approach a given bandwidth threshold to maximize the system performance. This trade-off has been explored in several research works [1], [2], [3].

On the other hand, with the aim of hiding, as much as possible, the huge memory latencies that current DRAM memories present, many commercial processors implement large Last Level Caches (LLC) and other microarchitectural mechanisms like prefetching or simultaneous multithreading. As an example, Figure 1 presents a memory hierarchy of the eight-core IBM Power 5 processor [4], which closely resembles the scheme followed by some processors like Intel Dunnington [5].

The latter processor supports the execution of two hardware threads per core. Therefore, a significant number of jobs can compete for accessing a low-level cache structure. For example, up to 8 processes can try to access the L3 cache in each quad-core. Other designs, like the quad-core Xeon presents a similar memory hierarchy with shared L2 caches [6]. Moreover, recent commercial designs [7] present large L3 shared caches with huge latencies (close to several tens of cycles) and can accommodate by around four to eight hardware threads per core. Therefore, the cache contention is a major design concern, which is expected to exacerbate in future microprocessor generations.

• *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*

In summary, current L2 and L3 caches are commonly shared by an increasing number of threads, thus bandwidth contention can appear at any level of the cache hierarchy. Therefore, these potential contention points must be tackled by the scheduler policy in order to maximize the system performance.

This paper has two main contributions. First, we characterize the performance sensitiveness of the set of benchmarks to each contention point in the memory hierarchy of a quad-core Intel Xeon X3320, showing that some benchmarks are even more sensitive to L2 cache contention than to main memory contention. Second, we propose two cache-hierarchy contention aware scheduling approaches for multi-core processors with shared caches. The first algorithm aims to prevent contention along the hierarchy by considering the bandwidth required by each process at each contention point. The second algorithm extends the first one by taking into account the degradation that benchmarks suffer when they are scheduled by memory contention-aware schedulers.

Despite that the processor studied in this work does not include multithreading capabilities and its cache hierarchy is relatively small when compared with the hierarchy of more recent processors, experimental results show that the proposal reaches performance improvements up to 9.56% in comparison with the Linux scheduler, while these benefits are always around or lower than 5% for an state-of-the-art memory-contention aware scheduler. Moreover, in some mixes the latter proposal triples the speedup achieved by the memory-contention aware scheduler.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the platform where the experiments are carried out. Section 4 presents the benchmark behavior and analyzes performance degradation due to both main memory and L2 contention. Section 5 introduces the scheduling proposals and Section 6 evaluates its performance. Finally, Section 7 presents some concluding remarks.

2 Related Work

Most research on bandwidth-aware multi-core schedulers focuses on mitigating the performance penalties due to either main memory contention [2], [3], [8], [1], [9] or LLC contention [10], [11], [12], [13], [14], [15], [16].

Regarding main memory contention, Antonopoulos et al. ([2], [3]) proposed several scheduling policies based on the memory bus bandwidth consumption of the processes running at the same time (from now on co-runners). In [2], the bus bandwidth consumption values are obtained by modifying the source code of the running applications, while in [3], less intrusive implementations based on processor performance information are explored. In both cases, the proposed policies try to match the total bandwidth requirements of the co-runners to the peak memory bus bandwidth. In a posterior work addressing SMP clusters [8], Koukis et al. consider the network bandwidth as well. Other works also address the trade-off between energy consumption and execution time taking into account the memory contention [9].

In a recent work, Xu et al. [1] prove that irregular memory access patterns can produce fine-grained contention when the required bandwidth is close to the peak bandwidth. To deal with this situation, they propose the use of the average bandwidth requirements of the applications instead of the whole available bandwidth. Authors estimate the Ideal Average Bandwidth (IABW) of a workload as the number of main memory accesses divided by the total execution time. In practice, the IABW is an approximation, since the exact average bandwidth consumption due to memory accesses depends on the final schedule. Therefore, the IABW is adjusted using polynomial regression methods.

Tang et al. [17] study the impact of sharing memory resources on datacenter applications. Authors analyze the impact of thread-to-core mapping, according to the memory behavior of the applications, when considering sharing memory resources. They found that there is both a sizable benefit and a potential degradation from improperly sharing resources. Authors present both a heuristic-based and an adaptive approach to enhance the thread-to-core assignment policies in the datacenter.

Regarding LLC contention, two orthogonal approaches are used: cache partitioning [10], [11], [12], [13] and cache-aware scheduling [14], [15], [16]. Cache partitioning mechanisms avoid cache starvation of the co-runners by implementing new hardware-based metrics and mechanisms that maximize throughput and/or improve fairness among co-runners. However, as pointed out by Sato et al. in [14] these mechanisms can severely limit the overall performance if applications with cache requirements exceeding the cache capacity are co-scheduled. On the other hand, Fedorova et al. [15], [16] show that contention-aware scheduling based on cache miss rate is effective and only requires accounting information already provided by hardware counters in modern microprocessors. However, they focus on scenarios where the number of jobs matches the number of cores, and hence, avoiding bandwidth contention by distributing bandwidth requirements along time is not considered.

In contrast to previous works, we propose a global solution that tackles the bandwidth contention problems that can arise at each level of the memory hierarchy. We only found in the literature one proposal by Kaseridis et al. [18] with such wide target. They rely on additional hardware based resource profilers and cache partitioning algorithms to avoid cache contention. However, unlike their work, we use existing hardware counters extensively and avoid contention points only by scheduling decisions. Thus, our solution does not require hardware modifications in existing platforms.

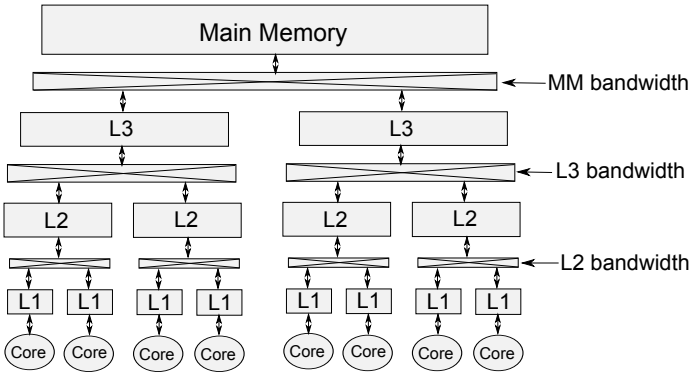

Fig. 1. Contention points related to the memory hierarchy of the IBM Power 5

3 Experimental Platform

The workload characterization is performed in a shared-memory quad-core Intel Xeon X3320 processor [6]. The processor has four cores without hyperthreading support and runs at 2.5 GHz with 4GB of DDR2 RAM.

The cache hierarchy of the quad-core consists of two 3MB L2 caches (LLC), each one shared by a pair of cores and private L1, 32KB for data and 32KB for instructions. It resembles the cache hierarchy presented in Figure 1, but with a two-level cache hierarchy. The main memory and each shared cache of the hierarchy are contention points since the structures of the lower level share the available bandwidth. As observed, the higher the number of cores and caches, the higher the number of contention points. Therefore, recent processors with deeper and wider memory hierarchies and future many-core processors should achieve better performance enhancements under cache-hierarchy contention aware schedulers.

The system runs a Fedora Core 10 Linux distribution with the kernel 2.6.29, which supports the monitoring software used in this work. This software, namely *perfmon2* [19], uses the library *libpfm* to access the hardware performance counters during processes execution and supports run-time measurement for co-running processes. Among the available statistics, the tool provides the number of cache misses for each cache structure.

4 Performance Degradation Analysis

This section analyzes the performance behavior of the SPEC CPU2006 benchmark suite [1]. First, we study the performance when running each benchmark alone in the experimental platform. Then, we analyze the performance degradation due to L2 contention and main memory contention. Finally, we measure the performance degradation caused by contention-aware schedulers.

1. Since the memory requirements of the mixes, which consist of 8 to 12 benchmarks, exceed the available physical memory, some benchmarks were launched with the train input data set, while those with low memory requirements and/or low execution time with train (just a few tens of quantums) were launched with the reference input data set.

To perform this study, each SPEC benchmark was concurrently launched with synthetic microbenchmarks, and, the number of executed cycles, instructions retired, L2 and L1 cache misses were measured. The microbenchmark is designed to inject synthetic traffic in the memory hierarchy and, depending on the requirements, it can mimic the behavior of either a main memory-bounded or L2-bounded application. Therefore, different microbenchmark configurations allow us to study different workload conditions. Microbenchmark design is described in Appendix I.

In addition to bandwidth, cache space also acts as an important contention point. Thus, both bandwidth contention and cache contention contribute to performance degradation. Nevertheless, the use of cache misses is also a good indicator as how contentious the cache usage is.

4.1 Benchmarks Characterization

In order to avoid interference from other co-runners, each benchmark was characterized running alone according to three main performance indexes: Instructions Per Cycle (IPC), Transaction Rate due to L1 misses (TR_{L1}), and Transaction Rate due to L2 misses (TR_{L2}) [2], both presented in transactions per microsecond. The transaction rate is used to refer to the number of transactions occurred over the memory system.

Figure 2 depicts the IPC for each integer and floating-point benchmark, while Figure 3 and Figure 4 show their TR_{L1} and TR_{L2}, respectively. A high correlation between the IPC and TR_{L2} is observed since the five benchmarks with the lowest IPC (*mcf, astar, milc, soplex* and *lbm*) present a relatively high TR_{L2} values. TR_{L1} has a less impact, although when this value surpasses 40 transactions per microsecond the IPC is always lower than 1 (*mcf, cactusADM, leslie3d, soples, GemsFDTD* and *lbm*), except for *libquantum* and *bwaves*.

A given benchmark can be classified as memory-bounded when its TR_{L2} is high enough to significantly increase main memory contention. In such a case, the

2. Although the overall Transaction Rate includes transactions other than cache misses (e.g. writebacks), this paper focuses in these ones since they are the most impacting on bandwidth consumption.

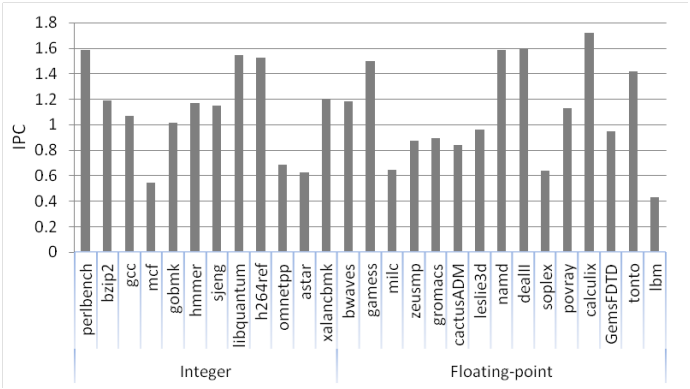

Fig. 2. IPC for each SPEC CPU2006 benchmark

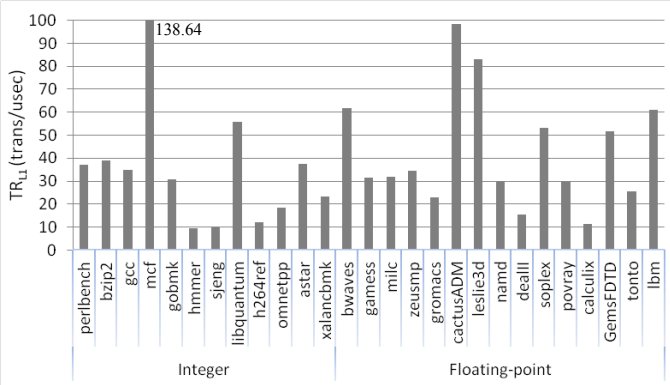

Fig. 3. TR_{L1} for each SPEC CPU2006 benchmark

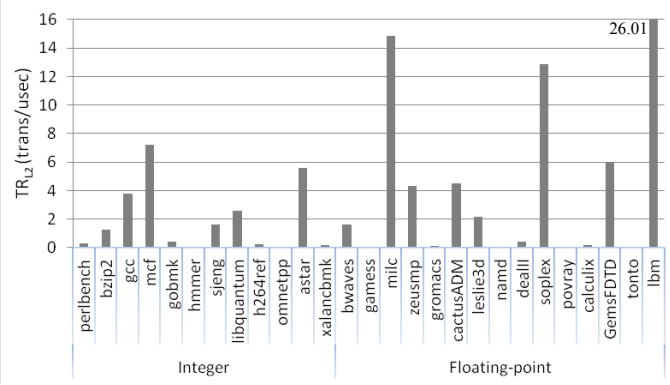

Fig. 4. TR_{L2} for each SPEC CPU2006 benchmark

benchmark will show a low IPC and will potentially affect the IPC of the co-runners. Likewise, a benchmark is considered to be L2-bounded when its TR_{L1} can cause L2 contention, which will affect the performance of those applications sharing the same L2. The effect of this type of contention is expected to grow in future many-core processors where the LLC cache structures are being shared by an increasing number of cores, most of them implementing multithreading capabilities. Note that L2-bounded does not necessarily means memory-bounded. This is the case of the *leslie3d* benchmark, with a TR_{L1} by about 80 trans/usec but a TR_{L2} around 2 trans/usec.

4.2 Degradation due to main memory contention

To check the performance degradation caused by main memory contention, we designed two experiments. The first experiment is aimed at checking the impact of the traffic created by the co-runners on the performance of a given benchmark. The second studies how the number co-runners and the core they are launched affect the performance of the benchmarks.

The first experiment was designed assuming that the system is fully loaded; that is, each core is busy running a process. To this end, each benchmark is concurrently launched with 3 memory-bounded instances of the microbenchmark. To explore the effects of having different traffic amounts, the microbenchmarks was configured to

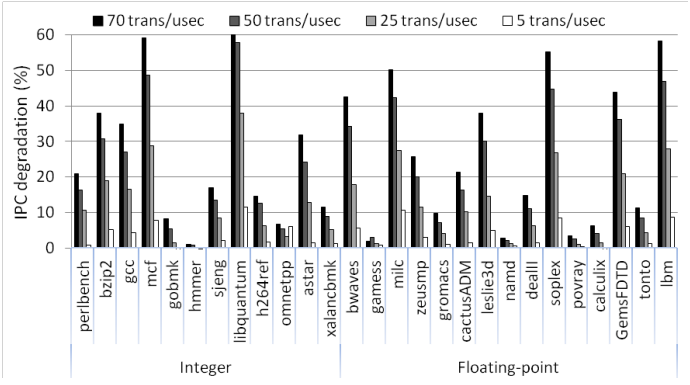

Fig. 5. IPC degradation due to main memory contention varying the TR_{L2} of the co-runners

obtain TR_{L2} values ranging from 5 to 70 trans/usec for each instance. The highest value of the range (i.e. 70) is the maximum value the microbenchmark can achieve in the experimental platform.

Figure 5 presents the results of this experiment. As observed, the amount of memory traffic generated by the microbenchmark can strongly affect the performance of the applications. In some cases, performance drops exceed 50%. This is the case of *mcf*, *libquantum*, *milc*, *soplex* and *lbm*, when the three instances of the microbenchmark are tuned to have a TR_{L2} equal to 70 transactions/microsecond. Few applications, like *hmmer*, *gamess*, *nand* or *povray*, are lightly affected since they show very low transaction rate between L2 and main memory. As expected, the lower the TR_{L2} of the microbenchmark the smaller the performance degradation. However, some benchmarks, like *libquantum*, *milc*, *soplex* and *lbm*, show important performance drops (greater than or close to 10%) even for an TR_{L2} of the microbenchmarks equal to

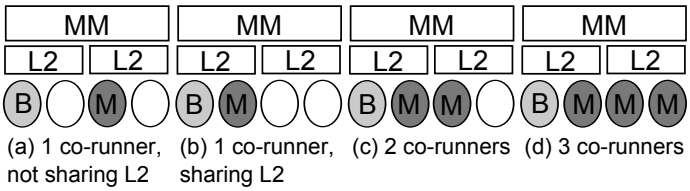

Fig. 6. Analyzed microbenchmarks scenarios

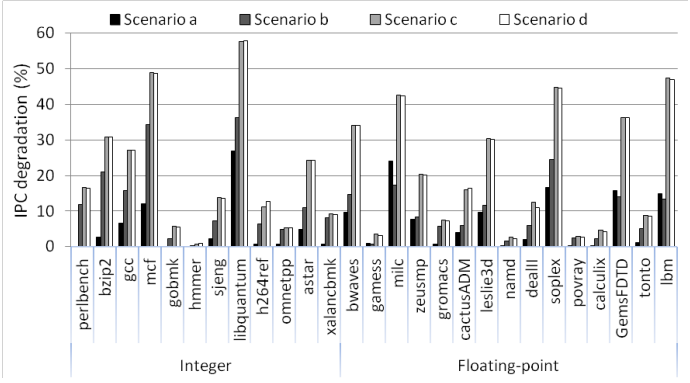

Fig. 7. IPC degradation due to main memory contention varying the execution context

5 trans/usec.

The second experiment varies the number of co-runners as well as the core in which they are executed. The microbenchmark instances were launched with a TR_{L2} equal to 50 trans/usec. Figure 6 shows the four scenarios analyzed and Figure 7 presents the results. Notice that scenario d experiences a performance degradation similar to that of scenario c. This means that memory is already saturated with 2 co-runners for almost all the studied benchmarks. Regarding the scenarios with one co-runner (a and b), most benchmarks suffer higher IPC degradation when the microbenchmark runs in a core that shares the LLC with the core running the benchmark (scenario b). Only a few memory-bounded benchmarks (*milc*, *GemsFDTD* and *lbm*) suffer higher degradation when the microbenchmark does not share the LLC (scenario a).

In shorts, some benchmarks suffer more degradation from the cache hierarchy contention (e.g. *mcf*, *libquantum* and *soplex*) while others (e.g. *milc*, *GemsFDTD* and *lbm*) are mainly affected by memory bandwidth contention. Therefore, it is critical to consider both when scheduling on machines with a complex memory hierarchy.

4.3 Degradation due to L2 Contention

To evaluate the performance degradation caused by L2 contention the microbenchmark parameters were tuned to stress the L2 cache but not the main memory, that is, L2 accesses will result in hits. Since each L2 cache is shared by a pair of cores, experiments focused only on a single L2 cache. Two processes were launched together, one SPEC benchmark and one L2-bounded instance of the microbenchmark. Hence, there was no benchmark running on the other pair of cores. We vary the induced TR_{L1} of the co-runner from 20 to 290 trans/usec, which is the maximum value reachable in the platform.

Figure 8 shows the results. As observed, the IPC of some benchmarks, like *mcf* and *soplex*, is strongly affected (IPC degradation is even higher than 10%) by the traffic created by other processes competing for the L2 cache. In addition, 12 benchmarks from 27 have a degradation higher than or close to 5% when they are co-scheduled with an L2-bounded instance of the microbenchmark with TR_{L1} equal to 290 trans/usec. This means that some benchmarks are highly sensitive to the L2 accesses of the co-runners. In fact, in some benchmarks like *bzip2*, *h264ref*, *omnetpp*, *xalancbmk* or *povray* the IPC degradation due to L2 contention can be higher than the caused by main memory contention, when the corresponding benchmark runs concurrently with one instance of the microbenchmark. For example, in *bzip2* the degradation caused by main memory contention when running concurrently with one memory-bounded microbenchmark is 2%, while one L2-bounded microbenchmark can degrade performance up to 5%.

Therefore, in this work we claim that, since the current industry trend is to increase the number of cores as well as their multithreading capabilities, a bandwitdh-aware scheduling policy for each level of the cache hierarchy can help the scheduler to improve the system performance.

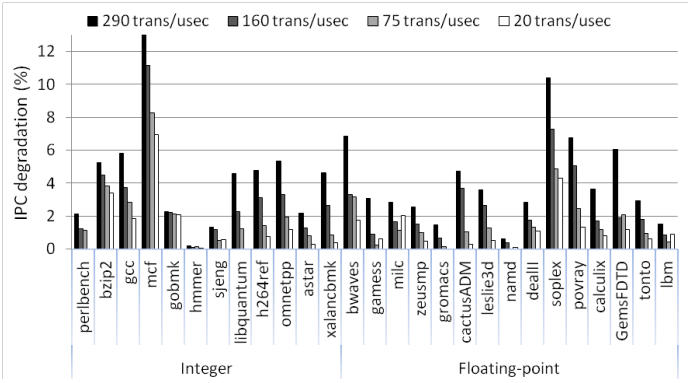

Fig. 8. IPC degradation due to L2 contention varying the TR_{L1} of the co-runners

4.4 Degradation due to contention aware scheduling

The last experiment analyzes the IPC degradation suffered by the benchmarks assuming a fixed main memory bandwidth utilization generated by all the processes running concurrently. The IPC degradation is evaluated for a bandwidth utilization of 30 transactions per microsecond, which is the average IABW of the evaluated mixes (see Appendix IV). This experiment reproduces the common situation created by state-of-the-art contention-aware schedulers, which try to achieve a constant bandwidth utilization as close as possible to the IABW. Therefore, the experiment obtains an IPC degradation that approaches to the suffered by each benchmark when it is executed under this kind of schedulers.

In order to simulate the described situation, the benchmarks are executed concurrently with three instances of the microbenchmark. The TR_{L2} of the microbenmark is tunned to reach an overall amount of 30 memory transactions per microsecond.

Figure 9 shows the results of this experiment. The observed degradation is highly correlated with the TR_{L2} presented by the benchmarks (Figure 4). Benchmarks with low TR_{L2} are not sensitive to the contention between L2 and main memory since their main memory accesses are not frequent. In fact, benchmarks with TR_{L2} lower than 2 trans/usec suffer an IPC degradation below 5% (except *dealII*, *bzip2* and *libquantum*, althougth the TR_{L2} of the last two is close to 2 trans/usec). In contrast, all the benchmarks with TR_{L2} above 2 trans/usec suffer a higher IPC degradation, which surpasses 10%, with the only exception of *astar*.

Depending on the degradation level, benchmarks can be classified in two categories. The *little sensitive* group includes the processes with an IPC degradation below 5%, which are little affected by the contention aware

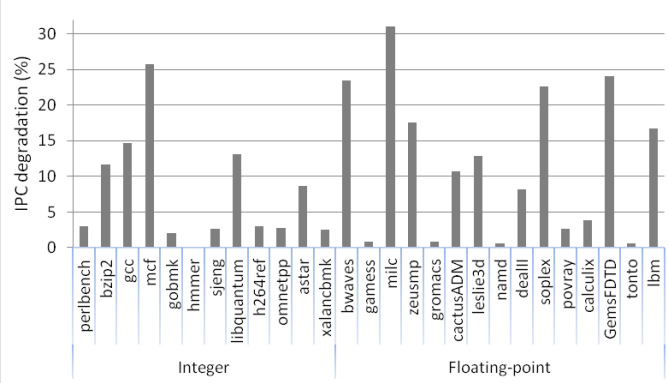

Fig. 9. IPC degradation with total BTR_{L2} of 30 trans/usec when running with three co-runners

scheduling. On the other hand, benchmarks with an IPC degradation between 5% and 35% are included in the *sensitive* category, since their degradation due to contention aware scheduling is higher. Both categories can be considered as bounded well since only two benchmarks present degradations between 5% and 10%.

The degradation observed in this experiment motivated us to design the scheduling algorithm proposed in Section 5.3. This scheduler uses the measured degradation to execute the processes with higher degradation in those execution periods with less main memory bandwidth requirements.

5 Cache Hierarchy Aware Scheduling

5.1 Baseline memory-contention aware scheduler

Numerous schedulers have been proposed dealing with main memory contention. Most proposals work as follows. First, they block the running processes, read the performance counters, and update the accounting of bandwidth requirements from the counter values. Then, the scheduler selects which processes will be executed concurrently during the next quantum according to their expected bandwidth utilization.

Typically, schedulers have pursued to keep full utilization of the available bandwidth, by selecting processes trying to match the peak memory bus bandwidth [3]. However, recent works proved that contention could exist before the bandwidth utilization reaches the peak bandwidth.

This work uses as baseline the scheduler proposed by Xu et al. [1], which defines an *Ideal Average Bandwidth* (IABW) that quantifies the main memory bandwidth demand of a workload. By scheduling jobs whose memory bandwidth requirements approach the IABW, performance degradation is reduced since bandwidth utilization is balanced along the workload execution time, so reducing contention.

5.2 Cache-hierarchy contention aware scheduler

The performance degradation analysis discussed above leads to the necessity of a job scheduling policy that is aware of the available bandwidth in each potential contention point of the cache hierarchy, and not only of the main memory bandwidth (as stated in previous proposals). Therefore, the scheduler must be aware of the cache transaction rates that each process experiences in any cache structure of the hierarchy.

The pseudocode of the cache-hierarchy contention aware scheduler is discussed in the Appendix II. The scheduler addresses the target bandwidth at each contention point and schedules the processes in n steps (as many as levels with at least two shared caches). The strategy follows a top-down approach, that is, in the first step processes are selected to match a target MM bandwidth (upper contention point in Figure 1). Then, the last level cache (LLC) bandwidth is addressed by balancing the transactions of caches in the immediately higher level. After that, contention points of the following levels with at least two shared caches of the memory hierarchy are addressed (if they exist). At the end, the jobs are allocated to concrete cores so that the bandwidth along the cache hierarchy is balanced. Notice that using cache bandwidth to guide the scheduling strategy also takes into consideration cache space contention implicitly.

5.3 IPC-degradation cache-hierarchy contention aware scheduler

To improve the scheduling performance, the benchmark characterization is used to provide useful information to enhance scheduling and allocation decisions. As mentioned above, the cache-hierarchy contention aware scheduler calculates the IABW of the mix and then, tries to schedule processes to approach the bandwidth utilization as close as possible to the IABW for the next quantum. The IPC-degradation cache-hierarchy contention aware scheduler uses the benchmark characterization (see Section 4.4), which classifies benchmarks as *sensitive* and *little sensitive*. This classification is used to estimate how the performance of the processes is degraded when scheduled by memory-contention aware schedulers. As observed in Figure 9, the performance degradation experienced by processes widely differs among them, so the scheduler can use this information to enhance the performance.

The key idea of this scheduling technique consists on favoring the performance of *sensitive* benchmarks. To this end, when a *sensitive* benchmark is selected to run during the next quantum, the scheduler selects its co-runners to reach an estimated main memory bandwidth consumption below the calculated IABW. To compensate this variation, *little sensitive* benchmarks will be scheduled to execute in situations where the total bandwidth consumed is above the IABW. Nevertheless, since *sensitive* processes are executed in favorable situations, a global performance enhancement is expected.

To include this technique in the scheduling algorithm, a *penalty* coefficient is used. This coefficient is defined as a proportional part of the IPC degradation suffered

Algorithm 1 IPC-degradation cache-hierarchy contention aware scheduler

Require: Benchmarks submitted with execution time, TR_{LLC} in stand alone execution.

1: $WIABW = \frac{\sum_{p=0}^{P}(TR_{LLC}^{p}+PenaltyCoef^{p})*T^{p}}{\frac{\sum_{p=0}^{P}T^{p}}{\#cores}}$
2: **while** there are unfinished jobs **do**
3: Block the executing processes and place them at the queue tail.
4: **for** each process P executed in the last quantum **do**
5: **for** each cache level L **do**
6: Update TR for process P in cache level L
7: **end for**
8: **end for**
9: $BW_{Remain} = WIABW$
10: Select the process P_head at the queue head.
11: $BW_{Remain} -= TR_{LLC}^{P\_head} + PenaltyCoef^{P\_head}$
12: $CPU_{Remain} = \#cores - 1$
13: **while** $CPU_{Remain} > 0$ **do**
14: select the process P that maximizes
15: $FITNESS(p) = \frac{1}{\left|\frac{BW_{Remain}}{CPU_{Remain}} - (BW_{required}^{P} + Penalty\ Coef^{P})\right|}$
16: $BW_{Remain} -= (TR_{LLC}^{P} + PenaltyCoef^{P})$, $CPU_{Remain}--$
17: **end while**
18: **for** each level *i* in the cache-hierarchy with shared caches beginning from the LLC **do**
19: $AVG_{TR}(L_{i-1}) = \frac{\sum TR_{L(i-1)}}{\#Caches\_at\_Li}$
20: **for** each cache in level L_i **do**
21: $BW_{Remain} = AVG_{TR}(L_{i-1})$, CPU_{Remain} = # cores sharing the cache
22: **while** $CPU_{Remain} > 0$ **do**
23: From the remaining processes selected to share the immediately lower memory level, select the process P that maximizes
24: $FITNESS(p) = \frac{1}{\left|\frac{BW_{Remain}}{CPU_{Remain}} - BW_{required}^{P}\right|}$
25: $BW_{Remain} -= BW_{required}^{P}$, $CPU_{Remain}--$
26: **end while**
27: **end for**
28: **end for**
29: Unblock the processes, and allocate them in the chosen cores.
30: Sleep during the quantum.
31: **end while**

by each benchmark. Different coefficient values were checked for performance, resulting the best penalty coefficient as a fifth of the process IPC degradation for *sensitive* benchmarks. Otherwise, zero is used as penalty coefficient. See Appendix III for further details.

Algorithm 1 presents the proposed algorithm considering the degradation due to contention aware scheduling. It extends the cache-hierarchy contention aware scheduler presented above. The approach can logically be seen as divided in a initialization step and three phases. In the initialization step the WIABW is calculated using the penalty coefficient of each process (line 1), which was not used to calculate the IABW.

In the first phase (lines 2 to 8) the scheduler blocks the running processes, updates their TR in each cache level and inserts them at the tail of the processes queue. To update the TRs of each process, the scheduler collects the required events using performance counters. These TR values will be used as the predicted TRs for the next execution quantum.

In the second phase (lines 9 to 17) the scheduler chooses the processes that will run during the next quantum according to their main memory bandwidth requirements. The process at the queue head is always selected to avoid process starvation, while the remaining ones are selected, until the number of cores is reached, according to the fitness function (line 15). The inclusion of the penalty coefficient causes the calculated WIABW of a mix to be higher than the IABW. When a *sensitive* process is selected, the penalty coefficient is subtracted (in addition to the TR_{LLC}) from the remaining bandwidth (line 16), so the remaining processes will achieve an overall TR_{LLC} lower than both WIABW and IABW, thus favoring its execution.

Finally, in the third phase (lines 18 to 28) the scheduler deals with the contention for the bandwidth located at the shared levels of the cache hierarchy (referred to as L3 bandwidth and L2 bandwidth in Figure 1) as the cache hierarchy aware scheduler does. The processes are assigned to each cache structure at each level according to their TR in that level so that the overall transactions at each cache level are balanced among the different cache structures.

In summary, the *sensitive* processes run in execution periods and with co-runners where the main memory transaction rate is lower favoring their performance. On the other hand, *little sensitive* processes run in scenarios with higher main memory transaction rate but they do not suffer a increase of their degradation for this situation.

6 Scheduler Evaluation

6.1 Evaluation methodology

To evaluate the effectiveness of the proposal we designed a set of ten mixes. Mixes 1 to 7 contain a number of benchmarks twice as large as the number of cores, while mixes 8 to 10 triple this value. Mix design is discussed in Appendix IV.

The execution time widely varies among the different benchmarks. To avoid that a scheduling policy prioritizes the longest jobs to provide the best performance in most mixes, we consider that each benchmark in the mix executes in the experiments as many instructions as it executes during a fixed amount of time running alone as done in [1]. In particular, we gathered the number of instructions that each benchmark runs during two minutes. When a benchmark is used in a experiment it is killed or relaunched (as many times as required) to complete the target number of instructions.

For evaluation purposes, we compared the performance of both scheduling proposals: i) cache-

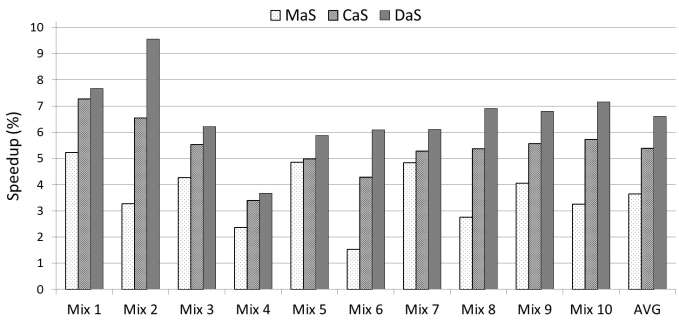

Fig. 10. Speedup over native Linux OS

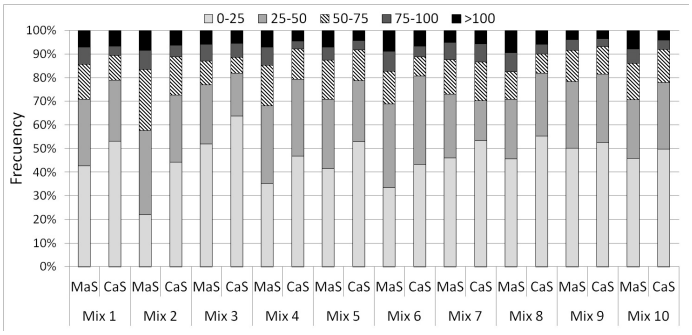

Fig. 11. TR_{L1} differences between the L2 shared caches

hierarchy contention aware scheduler (CaS), and ii) IPC-degradation cache-hierarchy contention aware scheduler (DaS) against two schedulers: a state-of-the-art memory-contention aware scheduler (MaS) and the Linux OS scheduler. The four schedulers were implemented as user-level schedulers, which run above the OS scheduler. They use Linux *ptrace_attach*, *ptrace_detach* and *sched_setaffinity* functions to force the OS to follow the desired scheduling. The studied schedulers share most of the code and overhead of managing processes and mainly differ in the process selection and allocation functions, so that the schedulers are fairly compared.

The implemented Linux scheduler selects all the processes to be run each quantum and leaves the native Linux OS scheduler to decide which processes are executed and the cores to which they are allocated. The MaS selects the processes to be executed according to their main memory bandwidth requirements, but does not allocate them to cores, which is a task performed by the OS scheduler. Finally, both CaS and DaS select the processes and allocate them to cores according to their respective algorithms. Quantum length was fixed to 200ms in the experiments.

6.2 Scheduler Performance

Figure 10 shows the speedup achieved by MaS and both proposed schedulers: CaS and DaS over the native Linux scheduler, considered as baseline. As observed, regardless of the benchmark, the proposals always provide better performance than the memory-contention aware scheduler. For MaS, the achieved speedup widely varies across mixes, ranging from 1.63% to 5.22%, with an average speedup of 3.61%, showing it can improve the performance of the mixes as stated in [1]. For CaS, the achieved speedup ranges from 3.38% to 7.26%, averaging 5.38%. These results show that a scheduler considering the contention across the memory hierarchy can improve the performance of a scheduler that only considers the main memory contention. The achieved speedup is improved by DaS, whose speedup ranges from 3.66% to 9.56%, averaging 6.64%. The average speedup achieved by DaS almost doubles the average speedup achieved by MaS. Furthermore, in half of the mixes (2, 6, 8, 9 and 10), DaS triples the speedup of MaS. The main reason behind the performance of CaS is that it balances the transactions among contention points along the cache hierarchy. Since the experimental platform has two shared L2 caches, the scheduler allocates jobs to cores taking into account that L1 misses must be balanced between both L2 caches. In this way, the L2 bandwidth contention is reduced, which turns into performance enhancements.

To estimate how well this balancing works, we measured the TR_{L1} affecting to both L2 caches and calculated their difference. Figure 11 presents the results. The histogram represents the frequency of the TR_{L1} difference between both L2 caches for MaS and CaS. Results are presented in intervals of 25 transactions per microsecond. The higher the frequency of the lower intervals (i.e. smaller difference) the better the TR is balanced between L2 caches.

For example, if we compare MaS bar versus CaS bar in mix 1, we can observe that for MaS, 40% of time (bottom bar) the TR_{L1} difference between both L2 caches is less than 25 transactions/us. The immediately upper bar indicates that by 30% of times the difference falls in the range [25-50] and so on. In contrast, for CaS, the [0-25] interval frequency increases up to 50% of time, resulting in better TR_{L1} distribution and better performance.

Results show a strong correlation between the frequency distribution and the speedup. For instance, mixes 2, 6, 8 and 10 present the widest distribution variation between both schedulers, which translates in the highest speedup variations. This can be appreciated in the lowest interval (i.e. 0-25) in mix 2, but also in the reduction of the intervals above 50 trans/usec in mixes 6, 8 and 10.

To provide a sound understanding of why TR balancing improves the performance, let's look inside the dynamic execution of a mix. In particular, let's focus on

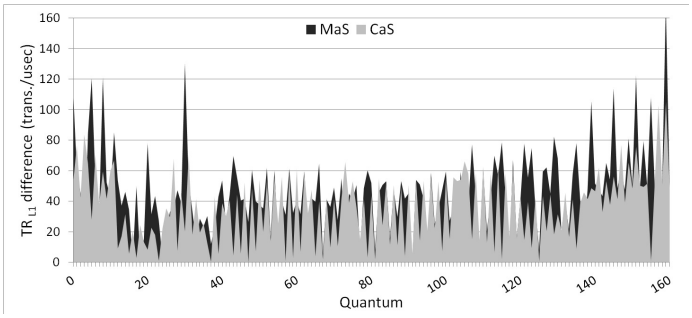

Fig. 12. TR_{L1} difference in the first 160 quanta

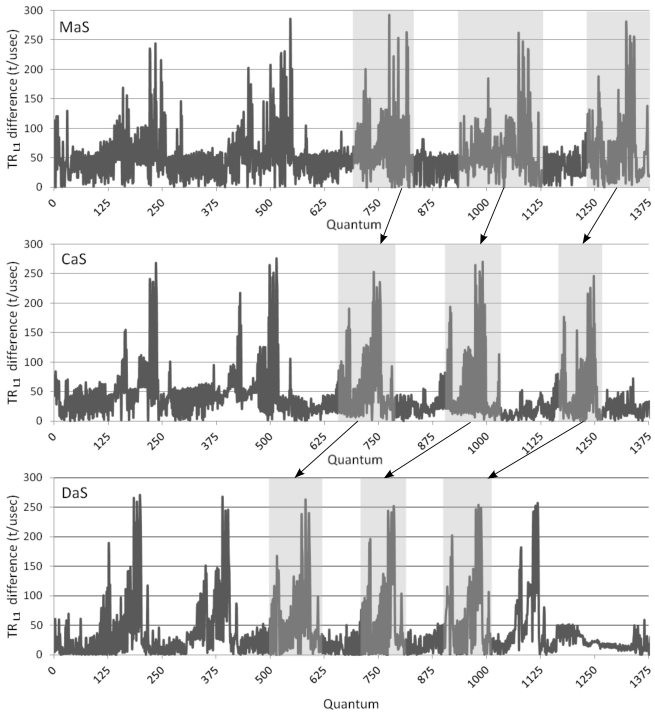

Fig. 13. TR_{L1} difference evolution with time

mix 2 where CaS improves by 50% the speedup achieved by MaS. Figure 12 shows the TR_{L1} difference of each quantum during the first 160 quanta of execution for both schedulers. The plot shows that the TR_{L1} difference for MaS is usually higher than for CaS. An even more important observation is that the peaks of this difference, which cause most of the contention are reduced by CaS, both in number and size.

Figure 13 presents the dynamic TR_{L1} differences using MaS, CaS and DaS during the first 1375 quanta of execution of mix 2. TR_{L1} differences, which are mainly caused by the *mcf* benchmark appear before in CaS than in MaS. Notice that this speedup is not achieved at expense of increasing the peak heights, since the heights are reduced too. This effect is improved by DaS that places the peaks ahead of CaS. Moreover, TR_{L1} differences between the peaks are also improved. Looking at the DaS plot, it can be appreciated that in many intervals the TR_{L1} difference falls always below 50 transactions/usec. Notice that the difference usually falls above this value in MaS.

Finally, to compare the benefits of DaS against CaS, we measured the percentage of benchmarks in each mix that reduce their execution time (speedup) and those that enlarge it (slowdown). Figure 14 shows the results. The first two intervals with negative values in the range refer to slowdown while the remaining ones (positive values) refer to benchmarks favored by DaS. As observed, 9 of 10 mixes are benefited for DaS. Moreover, 6 mixes present by 60% of their benchmarks favored by the DaS policy. The penalty coefficient included in DaS cause the *sensitive* benchmarks to execute in scenarios with less contention. Thus, these benchmarks present speedup. On the other hand, *little sensitive* benchmarks are executed

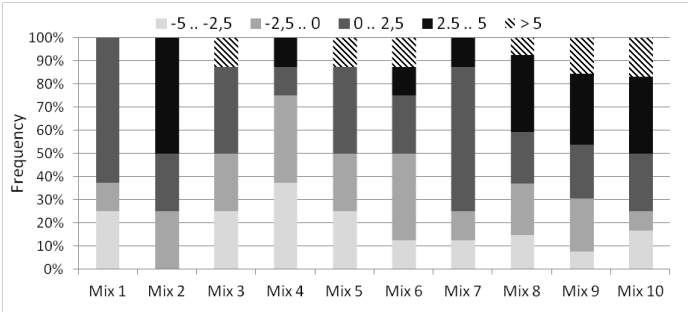

Fig. 14. Speedup of the benchmarks of each mix with DaS against CaS

in scenarios with higher bandwidth contention, slowing down their execution but with a lower impact on overall performance.

Mix 4 is the only mix where the percentage of benchmarks with slowdown is higher than the percentage of benchmarks with speedup. Even in this mix, the execution time using DaS is better than for CaS. Notice that the individual speedup of the benchmarks does not take into account the fact that at the end of the mix execution, some benchmarks can be executed when the number of processes is lower than the number of cores. When this situation is long enough, IPC of individual benchmarks is improved since there is less contention, but the mix execution time will not necessarily be improved.

7 Conclusions

This work has addressed the cache sharing contention in typical CMPs, and has proven that the system performance can drop due to bandwidth contention located at different levels of the memory hierarchy.

First, we have analyzed the benchmarks degradation on a commercial CMP processor due to bandwidth contention at the different contention points across the memory hierarchy. We found that, contrary to expected, and depending on the co-runners, some benchmarks are more sensitive to contention in higher levels of the memory hierarchy (e.g., shared L2) than to main memory contention.

These results lead us to claim that shared caches will increase their pressure in performance in future microprocessor generations. To deal with this performance concern, we have proposed two scheduling algorithms for generic CMPs. Both proposals do not require any additional hardware support, instead they only need the information provided by performance counters already available in current microprocessors.

The first proposed algorithm (cache-hierarchy contention aware scheduler) selects the processes taking into account the main memory bandwidth to reduce the global contention. As contention points can appear at each level of the memory hierarchy, the proposal follows a top-down multi-level approach that takes n steps (as many as shared cache levels) to plan a globally balanced schedule for the next quantum. The second

proposal (IPC-degradation cache-hierarchy contention aware scheduler) adds the studied degradation due to contention aware scheduling to the first algorithm. This degradation shows that some processes suffer a higher degradation assuming a fixed main memory bandwidth utilization generated by all the processes running concurrently. The algorithm favors the performance of the *sensitive* processes selecting the processes to run concurrently with a lower main memory bandwidth utilization.

Experimental results show that, compared to the native Linux scheduler, the achieved speedups range from 3.38% to 7.26% and from 3.66% to 9.56%, for the former and latter scheduler, respectively. The average speedup for the second algorithm is 6.64% and almost doubles the speedup achieved by a state-of-the-art memory-contention aware scheduler.

ACKNOWLEDGMENTS

This work was supported by the Spanish MINECO under Grant TIN2012-38341-C04-01, and by Universitat Politècnica de València under Grant PAID-05-12 SP20120748.

REFERENCES

[1] D. Xu, C. Wu, and P.-C. Yew, "On mitigating memory bandwidth contention through bandwidth-aware scheduling," in *PACT*, 2010, pp. 237–248.
[2] C. Antonopoulos, D. Nikolopoulos, and T. Papatheodorou, "Scheduling algorithms with bus bandwidth considerations for smps," in *Proc. of Parallel Processing*, 2003, pp. 547 –554.
[3] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou, "Realistic workload scheduling policies for taming the memory bandwidth bottleneck of smps," in *In Proc. of HiPC04*, pp. 286–296.
[4] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner, "Power5 system microarchitecture," *IBM J. Res. Dev.*, vol. 49, pp. 505–521, July 2005.
[5] R. Kuppuswamy, S. Sawant, S. Balasubramanian, P. Kaushik, N. Natarajan, and J. Gilbert, "Over one million tpcc with a 45nm 6-core xeon cpu," in *IEEE International Solid-State Circuits Conference*, 2009, pp. 70 –71,71a.
[6] G. Varghese, J. Sanjeev, T. Chao, S. Ken, D. Satish, S. Scott, N. Ves, K. Tanveer, S. Sanjib, and S. Puneet, "Penryn: 45-nm next generation intel core 2 processor," in *IEEE Asian Solid-State Circuits Conference*, 2007, pp. 14–17.
[7] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd, "Power7: IBM's next-generation server processor," *IEEE Micro*, vol. 30, pp. 7–15, 2010.
[8] E. Koukis and N. Koziris, "Memory and network bandwidth aware scheduling of multiprogrammed workloads on clusters of smps," in *Proc. of ICPADS'06*, pp. 345–354.
[9] F. Pinel, J. E. Pecero, P. Bouvry, and S. U. Khan, "Memory-aware green scheduling on multi-core processors," in *Proc. of ICPPW '10*, pp. 485–488.
[10] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. MICRO 39*, 2006, pp. 423–432.
[11] G. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proc. of HPCA'02*, pp. 117 – 128.
[12] J. Chang and G. S. Sohi, "Cooperative cache partitioning for chip multiprocessors," in *Proc. of ICS '07*, pp. 242–252.
[13] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proc. of PACT '04*, 2004, pp. 111–122.
[14] M. Sato, I. Kotera, R. Egawa, H. Takizawa, and H. Kobayashi, "A cache-aware thread scheduling policy for multi-core processors," in *International Conference on PDCN'09*, pp. 109 –114.
[15] A. Fedorova, S. Blagodurov, and S. Zhuravlev, "Managing contention for shared resources on multicore processors," *Commun. ACM*, vol. 53, pp. 49–57, 2010.
[16] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proc. of ASPLOS '10*, pp. 129–142.
[17] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M.-L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *ISCA'11*.
[18] D. Kaseridis, J. Stuecheli, J. Chen, and L. John, "A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large cmp systems," in *IEEE International Symposium on HPCA'10*, pp. 1 –11.
[19] S. Jarp, R. Jurga, and A. Nowak, "Perfmon2: a leap forward in performance monitoring," *Journal of Physics: Conference Series*, vol. 119, no. 4, p. 042017, 2008.

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

Salvador Petit received the PhD degree in computer engineering from the Universitat Politècnica de València, Spain. Currently, he is an associate professor in the Department of Computer Engineering at the UPV where he has several courses on computer organization. His research topics include multithreaded and multicore processors, memory hierarchy design, as well as real-time systems. He is a member of the IEEE and the IEEE Computer Society.

Julio Sahuquillo received his BS, MS, and PhD degrees in Computer Engineering from the Universidad Politécnica de Valencia (Spain). Since 2002 he is an Associate Professor at the Department of Computer Engineering. He has taught several courses on computer organization and architecture. He has published more than 100 refereed conference and journal papers. His current research topics include multi- and manycore processors, memory hierarchy design, cache coherence, and power dissipation. He has cochaired several Workshops related with these topics, collocated in conjunction with IEEE supported conferences. He is a member of the IEEE Computer Society.

José Duato received the MS and PhD degrees in electrical engineering from the Universitat Politècnica de València, Spain, in 1981 and 1985, respectively. He was an adjunct professor with the Department of Computer and Information Science, The Ohio State University, Columbus. He is currently a professor with the Department of Computer Engineering (DISCA) at the UPV. His research interests include interconnection networks and multiprocessor architectures. 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. He is the first author of *Interconnection Networks: An Engineering Approach*.He was a member of the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, and *IEEE Computer Architecture Letters*. He was cochair, member of the steering committee, vice-chair, or member of the program committee in more than 55 conferences, including the most prestigious conferences in his area: HPCA, ISCA, IPDPS, ICPP, EuroPar, and HiPC.