Understanding Cache Hierarchy Contention in CMPs to Improve Job Scheduling

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

***Abstract*—In order to improve CMP performance, recent research has focused on scheduling to mitigate contention produced by the limited memory bandwidth. Nowadays, commercial CMPs implement multi-level cache hierarchies where last level caches are shared by at least two cache structures located at the immediately lower cache level. In turn, these caches can be 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.**

In this paper we characterize the impact on performance of the different contention points that appear along the memory subsystem. Then, we propose a generic scheduling strategy for CMPs that takes into account the available bandwidth at each level of the cache hierarchy. The proposed strategy selects the processes to be co-scheduled and allocates them to cores in order to minimize contention effects.

The proposal has been implemented and evaluated in a commercial single-threaded quad-core processor with a relatively small two-level cache hierarchy. Despite these potential contention limitations are less than in recent processor designs, compared to the Linux scheduler, the proposal reaches performance improvements up to 9% while these benefits (across the studied benchmark mixes) are always lower than 6% for a memory-aware scheduler that does not take into account the cache hierarchy. Moreover, in some cases the proposal doubles the speedup achieved by the memory-aware scheduler.

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

I. 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 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-aware strategies can help the scheduler to reduce memory contention by avoiding the concurrent execution of memory-hungry applications. These strategies take into account the total bandwidth required by applications and schedule a set of them to execute concurrently, whose cumulated bandwidth requirements does not exceed the available bandwidth. Otherwise, performance could severely be damaged due to memory contention. Nevertheless, previous research has shown that the scheduler must try to approach to 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 multithreading. As an example, Figure 7 presents a memory hierarchy interconnecting two quad-core chips, which closely resembles the scheme followed by the IBM Power 5 [4]. This processor supports the execution of two hardware threads per core. Therefore, a significant number of jobs can compete for accessing to a given shared 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, used to carry out the experiments in this paper, present a similar memory hierarchy with shared L2 caches [5]. Moreover, recent commercial designs [6], [7] present larger shared caches with huge L3 latencies (close to one hundred 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.

In summary, current L2 and L3 caches use to be shared by an increasing number of threads, thus memory 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 charac-

terize the performance sensitiveness of the set of benchmarks to each contention point in the memory hierarchy of a quad-core Intel Xeon, showing that some benchmarks are even more sensitive to L2 cache contention than to main memory contention. Second, we propose a scheduling approach for multi-core processors with shared caches. The proposed approach takes n steps (as many as cache levels) to schedule the jobs, and follows a top-down strategy, from the lowest to the highest shared cache level of the cache hierarchy.

Despite that the cache hierarchy is relatively smaller compared to some existing processors, and that cores do not include multithreaded capabilities; experimental results show that the proposal reaches performance improvements up to 9% in comparison with the Linux scheduler, while these benefits are always lower than 6% for a memory-aware scheduler that does not take into account the cache hierarchy. Moreover, in some cases the proposal doubles the speedup achieved by the memory-aware scheduler.

The rest of this paper is organized as follows. Section II discusses related work. Section III describes the platform where the experiments are carried out. Section IV presents the benchmark behavior and analyzes performance degradation due to both memory and L2 contention. Section V introduces the proposal and Section VI evaluates its performance. Finaly, Section VII presents some concluding remarks.

II. Related Work

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

Regarding memory contention, Antonopoulos et al. ([2], [3]) proposed several scheduling policies based on the memory bus bandwidth consumption of the co-runners. In [2], the information about this consumption is 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 [8] addressing SMP clusters, Koukis et al. take into account 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 running applications instead of the whole available bandwidth. Authors estimate the Ideal Average BandWidth (IABW) of a workload as the total number of LLC misses divided by the total execution time. In practice, the IABW is an approximation, since the exact average bandwidth consumption due to LLC misses depends on the final schedule. Therefore, the IABW is adjusted using polynomial regression methods to obtain the optimal scheduling bandwidth.

Regarding LLC contention, two complementary 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. Therefore, the scheduler must be aware of cache to avoid this type of situation. 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.

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. [17] with such wide target. In their work, they rely on additional hardware based resource profilers and cache partitioning algorithms to avoid cache contention. However, unlike this 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.

III. Experimental Platform

Workload characterization was performed in a shared-memory quad-core Intel Xeon X3320 processor [5], with the hardware specification summarized in Table I. The cache hierarchy of the quad-core consists of two 3MB L2 caches (LLC), each one shared by a pair of cores. The proposed scheme was implemented and evaluated in a user-level process manager using the ptrace Linux system call [18].

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

Most current processors implement performance counters for debugging purposes, which keep track of the number of cycles and events like the number of instructions executed and misses in each cache level. Apart of these counters, the proposed algorithm has no extra hardware requirement.

CPU	Intel Xeon X3320
Frequency	2.5 GHz
Number of cores	4
Multithreading	No
L1 cache	Code L1: 4 x 32 KB Data L1: 4 x 32 KB
L2 cache	2 x 3 MB shared, 12-way
Memory	4GB (2GB x 2) DDR2

Table I
SYSTEM SPECIFICATION

The events gathered by these counters usually differ among commercial processors. The major requirement is that the system must be able to collect the information in a per process basis, as done by Linux OS using the *libpfm* library.

IV. PERFORMANCE DEGRADATION ANALYSIS

This section explores the performance of the entire SPEC CPU2006 benchmark suite. First, we study the performance behavior when running each benchmark alone in the experimental platform. Then, we analyze the performance degradation due to L2 contention and memory contention. To this end, each SPEC benchmark was concurrently launched with synthetic benchmarks, measuring the unhalted core cycles, instructions retired, L2 cache misses (LLC) and L1 cache misses with the *pfmon2* tool.

A. Benchmarks Characterization

Each benchmark was characterized running alone according to three main performance indexes: IPC, MPKI (Misses Per Kilo Instructions) in the L1 cache, and MPKI in the L2 cache. In this way, interferences of other benchmarks are avoided. The latter two indexes will be referred to as MPKI_{L1} and MPKI_{L2}, respectively.

Figure 1 depicts the IPC for each integer and floating-point benchmark. Figure 2 and Figure 3 show the MPKI_{L1} and MPKI_{L2}, respectively. As expected, a high MPKI value results in a low IPC. This is the case of *mcf*, *gobmk*, *astar*, *specrand*, and *lbm* applications. An interesting observation is that, mainly in integer applications, there is a correlation between MPKI_{L1} and MPKI_{L2}. For example, the three integer benchmarks with MPKI_{L1} greater than 20 present the highest MPKI_{L2} (higher than 1.5).

Depending on the MPKI results, a given benchmark can be classified as memory-bounded when its MPKI_{L2} is high enough to increase significantly the memory contention. In such a case, the 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 MPKI_{L1} can cause L2 contention, which will affect the performance of those applications accessing the shared L2. Note that L2-bounded does not necessarily means memory-bounded. This is the case of the $leslie3d$ benchmark, with an MPKI_{L1} by about 35 but an MPKI_{L2} around 2. Note that the effect of the latter 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.

B. Degradation due to Memory Contention

To analyze the performance degradation that main memory contention causes in a given benchmark, we designed a memory-hungry synthetic microbenchmark, which is used as co-runner. This microbenchmark creates contention by injecting synthetic traffic in an infinite loop. To parametrize the induced contention, the microbenchmark includes an argument specifying the number of *nop* instructions that each iteration of the loop executes.

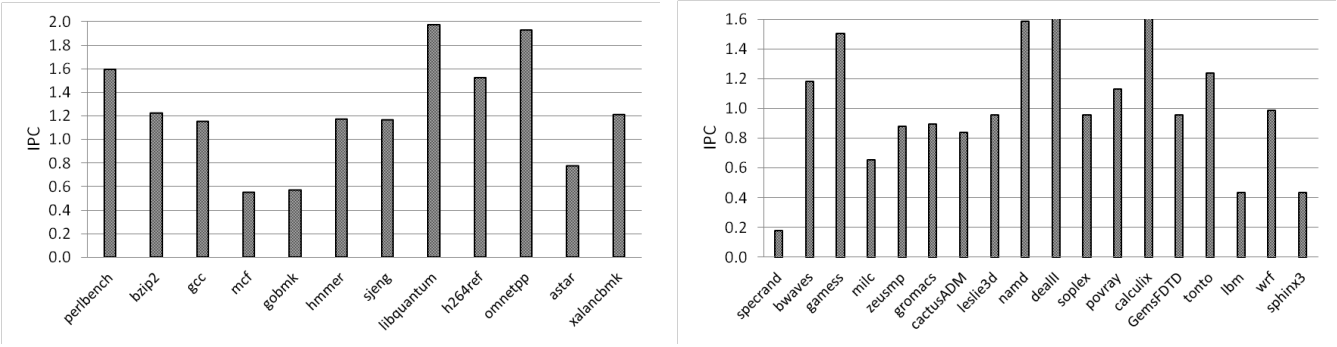
The designed microbenchmark can mimic the behavior of either a memory-bounded application or an L2-bounded application. Each iteration of this program executes memory instructions that miss in a given target L_i level of the memory hierarchy and hit in the next L_{i+1} level. Thus, depending on the target level (i.e., L1 or L2 in the experimental platform), the benchmark enables the study of how contention in the next level affects the performance of a given application. To sum up, the microbenchmark is used to evaluate how the IPC of a given benchmark degrades due to either memory contention or cache contention.

Listing 1 presents the core loop of the microbenchmark code. Parameters N and CACHE_LINE_SIZE refer to the number of lines and the line size of a given cache organization. In order to force continuous misses, these parameters must be properly tuned according to the target cache.

Listing 1. Microbenchmark code

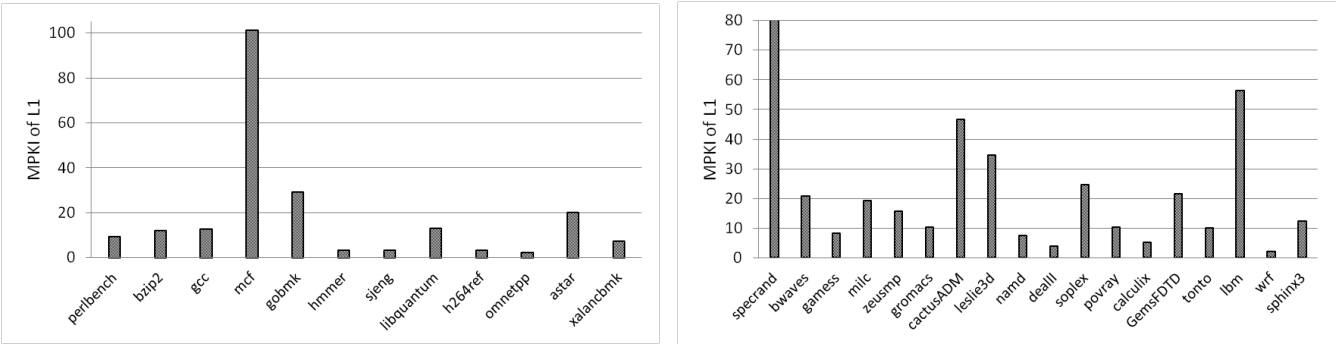
```
int A[N][CACHE_LINE_SIZE];
int B[N][CACHE_LINE_SIZE];
while (1) {
    for (i=0; i< (#misses/2); i++)
    {
        A[i][0] = B[i][0];
    }
    for (i=0; i< (#nops; i++) {
        asm("nop");
    }
}
```

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



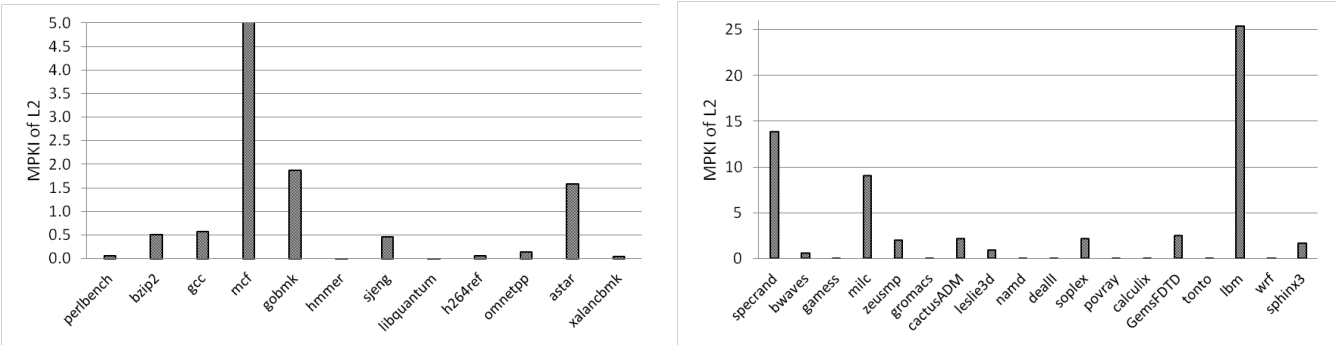
(a) integer (b) floating-point

Figure 1. IPC for each SPEC CPU2006 benchmark



(a) integer (b) floating-point

Figure 2. L1 MPKI for each SPEC CPU2006 benchmark



(a) integer (b) floating-point

Figure 3. L2 MPKI for each SPEC CPU2006 benchmark

The first experiment was designed assuming that the system is fully loaded; that is, each core is busy running a program. To this end, each benchmark is concurrently launched with 3 memory-bounded instances of the microbenchmark [1]. To explore the effects of having different traffic amounts, the number of *nop* instructions executed in each loop iteration was configured to obtain MPKI values ranging from 3 to 42 for each microbenchmark instance. The highest range value (i.e., 42) is the maximum value the microbenchmark can achieve in the experimental platform, that is, when the number of *nops* is set to 0. In such a case, this MPKI value translates to around 27 transactions per μs between the L2 cache and main memory.

Figure 4 presents the results of this experiment. As observed, the amount of memory traffic generated by the

[1]each instance accesses 25% the cache space of its core (i.e. 0.75 MB)

microbenchmark can strongly affect the performance of the applications. In some cases, and for an MPKI equal to 42, performance drops are as large as 60%. This is the case of *gobmk*, *milc* and *lbm*, when the three instances of the microbenchmark are tuned to have an MPKI equal to 42 in the L2 cache. A few applications, like *hmmer*, are poorly affected since they show very low MPKI both in L1 cache and L2 caches. As expected, the lower the MPKI of the microbenchmark the smaller the performance degradation. However, some applications, like *gobmk*, show important performance drops (larger than 30%) even for an MPKI equal to 3.

The second experiment varies the number of co-runners as well as the core in which they are executed. In the Intel Xeon, core 0 and core 2 refer to the pair of cores sharing one of the L2 caches, and core 1 and core 3 refer to the pair sharing the other L2. In this experiment, SPEC benchmarks always run in core 0, and we vary the cores where the instances of the microbenchmark were ran. The MPKI parameter of all the microbenchmark instances is set to 42. Four different situations were analyzed:

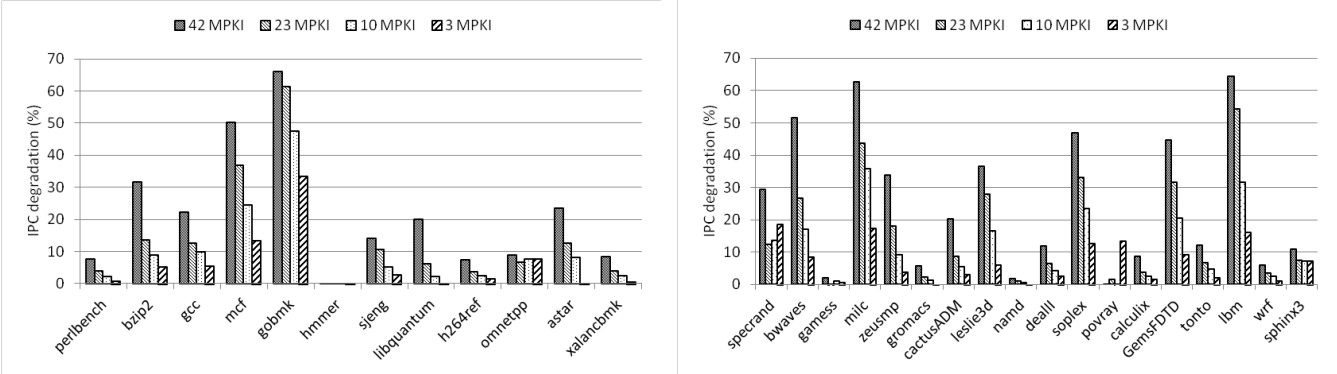
- Mem-b c1: the microbenchmark runs in core 1 so both applications (benchmark and microbenchmark) access to different L2 caches.
- Mem-b c2: the microbenchmark runs in core 2 so both applications compete for accessing the same L2 cache.
- Mem-b c1-2: two instances of the microbenchmark are executed, one in core 2 and one in core 1.
- Mem-b c1-2-3: three instances of the microbenchmark are executed. This case corresponds to the first column of the results of the previous experiment.

Figure 5 shows the results. As observed, some benchmarks are less sensitive to the pair in which the co-runner is launched. This happens because in these benchmarks, memory contention becomes the major performance bottleneck. The expected behavior is that, if the co-runners run in the other pair, then memory will be more frequently accessed so hurting the performance more severely. This is the common case (e.g., *bzip2*, *gcc*, or *mcf*). However, there are some benchmarks, like *gobmk*, with the opposite behavior. The reason is that when the co-runner runs in the same pair, it brings more memory blocks to the shared L2 cache (one per each memory access). Therefore, the microbenchmark is competing for L2 space, so increasing the miss rate of the tested benchmark in that cache. Finally, notice that the case with the highest memory requirements (i.e., Mem-b c1-2-3) experiences similar performance degradation as Mem-b c1-2. This is due to the memory being already saturated with 3 co-runners, thus the number of BTR_{L2} is similar in both cases.

C. Degradation due to L2 Contention

To evaluate the performance degradation caused by L2 contention, a similar set of experiments was performed. To this end, the microbenchmark parameters were tuned to stress the shared L2 cache but not the main memory. In this experiment the microbenchmark always hits the shared L2 cache. Since each L2 cache is shared by a pair of cores, experiments focused only on a single L2 cache. Two benchmarks 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 MPKI_{L1} of the co-runner from 10 to 132, which is the maximum value reachable in the platform when the number of *nops* is set to zero.

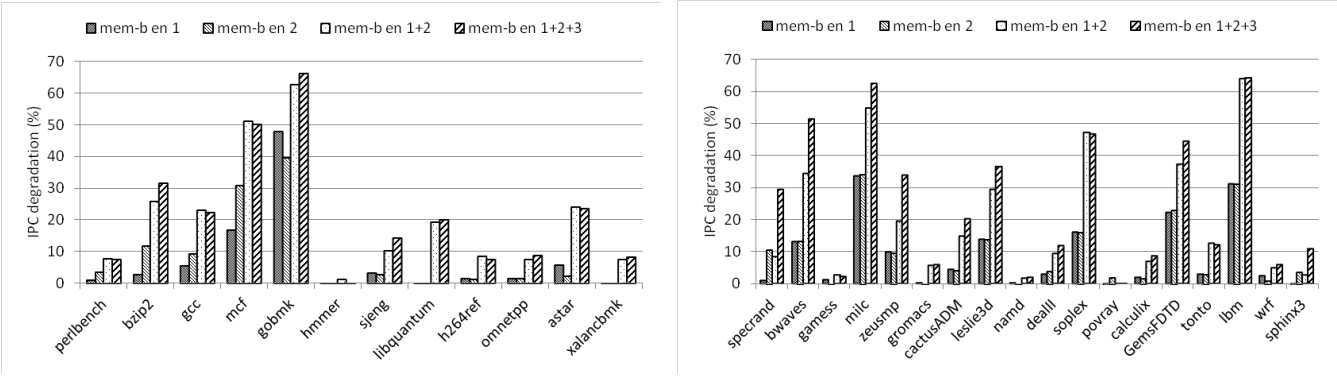
Figure 6 shows the results. As observed, the IPC of some benchmarks, like *mcf*, *specrand* and *soplex*, are strongly affected (IPC degradation is even higher than 10%) by the traffic created by others processes competing for the L2 cache. In addition, 12 benchmarks from 28 have a degradation higher than or close to 5% when they are co-scheduled with an L2-bounded instance of the microbenchmark with MPKI_{L1} equal to 133. This means that some benchmarks are highly sensitive to the L2 accesses of the co-runners. In



(a) integer

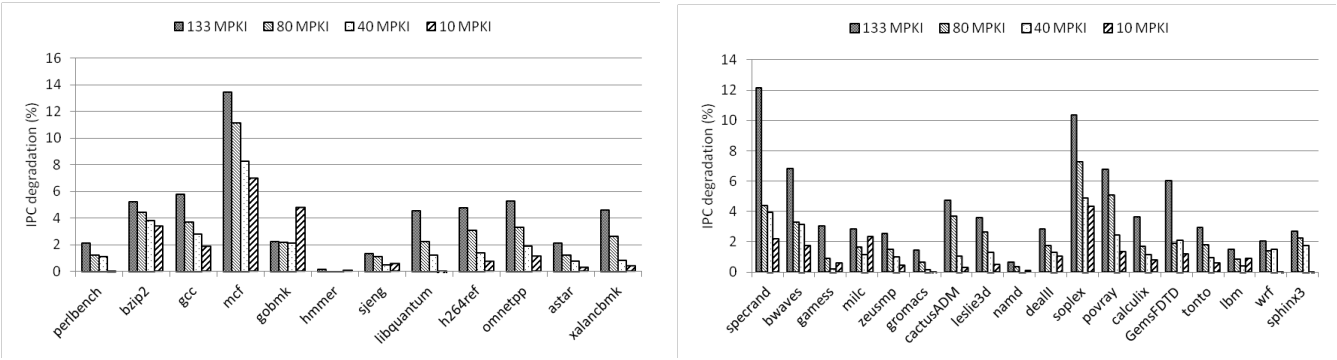
(b) floating-point

Figure 4. IPC degradation due to memory contention varying the mpki of co-runners



(a) integer (b) floating-point

Figure 5. IPC degradation due to memory contention varying the number of co-runners



(a) integer (b) floating-point

Figure 6. IPC degradation due to L2 contention

other words, in some benchmarks and, depending on the co-runners, IPC degradation due to L2 contention can be similar or even higher than the caused by memory contention.

As expected, the IPC degradation is lower than the $MPKI_{L1}$ of the co-runner decreases. However, even for an $MPKI_{L1}$ equal to 10, we can observe noticeable degradations (around 7%) like the experienced by *mcf*. Therefore, in this paper we claim that, since the current industry trend is to increase the number of cores as well as their multithreading capabilities, an scheduling policy aware of the bandwidth requirements for each level of the cache hierarchy can help the scheduler to improve the system performance.

V. Cache Hierarchy Aware Scheduling

A. Baseline memory aware scheduling

Numerous schedulers have been proposed to deal with memory contention. Most of the proposals work as follows: block the running processes, read the performance counters, and update the bandwidth requirements from the counter values. Then, the scheduler is responsible for selecting the processes to be executed concurrently during the next quantum attending 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 achieves 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 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.

B. Proposed cache-hierarchy aware scheduler

The performance degradation analysis discussed above leads to the necessity of a job scheduling that is aware of the available bandwidth in each potential contention point of the cache hierarchy, and not only of the memory bandwidth (as stated in previous proposals). For this purpose, the scheduler must know the cache misses that each process experiences in any cache structure of the hierarchy. As mentioned above, in this work these values were captured with the *perfmon2*

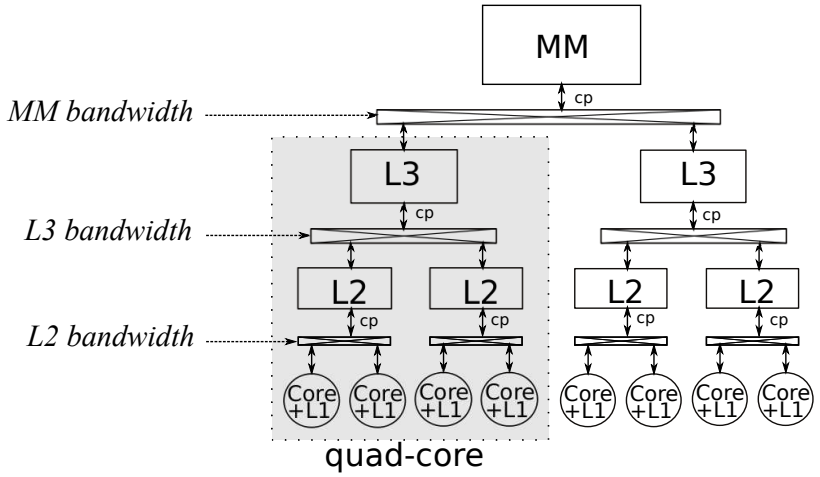


Figure 7. Contention points related to the memory subsystem in a three-level cache hierarchy CMP

monitor.

The BTR (bus transaction rate) has been typically used to refer to the number of transactions occurred over the memory system bus. This variable depends on the number of misses of the last level cache (L3 in Figure 7). Since the focus of this work is on the bandwidth requirements for each level of the memory subsystem, we use the terms BTR_{MM}, BTR_{L3}, and BTR_{L2} to refer to the BTR of main memory, L3, and L2, respectively. On the other hand, if the processor implements multithreading capabilities, the L1 cache must be also considered since it behaves as a shared memory structure. Notice that to quantify the BTR for a given cache level, that is, the bandwidth requirements of the running processes, we need to measure the misses that those processes experience in the immediately lower cache level. For instance, the BTR of each L3 cache is estimated with the misses that the processes have in the L2 caches sharing that L3. The BTR values were calculated from the values gathered in the performance counters, using the events *number of misses of the cache* structure and *unhalted core cycles* of each process.

The proposed scheduler addresses the target bandwidth at each contention point and schedules the jobs in n steps (as many as cache levels). The strategy follows a top-down approach, that is, in the first step jobs are selected to match a target MM bandwidth (upper contention point in Figure 7). Then, the last level cache (LLC) bandwidth is addressed by balancing the BTR of caches in the immediately higher level. After that, contention points of the following levels of the cache hierarchy are addressed. Finally, the last step allocates jobs to cores.

Algorithm 1 presents the proposed approach. In the first part the scheduler updates the BTR values. Each quantum, the algorithm gathers the BTR values experienced for every process in each cache level. The values gathered during a given quantum are used by the scheduler as the predicted BTR for the next quantum. In particular, in our experimental platform the BTR of both memory and shared L2 caches are updated for every process with the gathered values during the last quantum.

Algorithm 1 Cache-hierarchy memory aware scheduler

```
Block the executing processes and place them at the queue tail.
for each process P executed in the last quantum do
  for each cache level L do
    Update BTR for process P in cache level L
  end for
end for
while there are unfinished jobs do
  BW_Remain = average memory bandwidth
  Select the process at the queue head and update BW_Remain
  while selected processes < cores do
    select the process that maximizes
    FITNESS(p) = 1 / |BW_Remain/CPU_Remain − BW^P_required|
    and update BW_Remain
  end while
  for i = max_cache_level downto level 2 do
    AVG_BTR(L_i) = Σ BTR of L(i−1) / #Caches at Li
    for each cache in level L_i do
      BW_Remain = AVG_BTR(L_i)
      while #selected processes for the cache < # cores
      sharing the cache do
        Select the process that maximizes the FITNESS(p)
        function and update BW_Remain
      end while
    end for
  end for
  Unblock the processes, and allocate them in the chosen core.
end while
```

The main loop of the algorithm consists of a while loop that selects the jobs to be scheduled the next quantum. The first sentence of the loop estimates the target bandwidth BW_{Remain}. This parameter is initialized to the average

BTR_{MM} of all the processes of the mix weighted by their execution time. The resulting value is then slightly reduced to enhance the bus transaction rate [1].

The processes still pending to finish their execution are kept in a software queue structure. When a quantum expires, the processes executed during this quantum are placed at the queue tail. For the next quantum, the proposed algorithm always schedules the process located at the queue head to avoid process starvation, and then, selects those that maximize the fitness function described in [2], [3].

Each time a process is selected for execution, BW_{Remain} must be updated accordingly.

The result of this selection step is the list of processes to be executed taking into account the MM bandwidth constraint; that is, the upper contention point represented in Figure 7. Note that the number of processes selected is limited by the number of available cores (cores are assumed to be single-threaded). For example, eight processes will be selected for the two quad-core of the figure.

After that, the algorithm deals with the contention points located at the shared levels of the cache hierarchy; referred to as L3 bandwidth and L2 bandwidth in Figure 7. To this end, for each level L_i, the required BTR of all caches at L_{i-1} level is estimated, and averaged considering the number of caches at L_i level. Then, processes for each cache structure at L_i level are selected according to the fitness function. Notice that in the last iteration, when the loop reaches the lowest shared cache level (i.e., L2 in the example) the algorithm selects two processes for each L2 cache, which are subsequently allocated in any of the two cores sharing the L2.

Mixes	Benchmarks
Mix 1	*GemsFDTD, H264ref, Hmmer, Lbm, Lbm, Mcf, Tonto, Xalancbmk*
Mix 2	*Astar, GemsFDTD, H264ref, Hmmer, Lbm, Mcf, Tonto, Xalancbmk*
Mix 3	*Astar, GemsFDTD, Hmmer, Lbm, Lbm, Mcf, Tonto, Xalancbmk*
Mix 4	*Astar, CactusADM, GemsFDTD, Lbm, Lbm, Mcf, Tonto, Xalancbmk*
Mix 5	*Astar, Bwaves, CactusADM, Lbm, GemsFDTD, Mcf, Tonto, Xalancbmk*
Mix 6	*Astar, DealII, GemsFDTD, H264ref, Lbm, Mcf, Namd, Sjeng*
Mix 7	*Astar, Bwaves, DealII, H264ref, Lbm, Lbm, Mcf, Sjeng*
Mix 8	*CactusADM, GemsFDTD, Mcf, Milc, Lbm, Leslie3d, Tonto, ZeusMP*
Mix 9	*Bwaves, CactusADM, GemsFDTD, GemsFDTD,Lbm, Mcf, Milc, Povray*
Mix 10	*Astar, CactusADM, Gromacs, Lbm Leslie3d, Mcf, Mcf, Namd*

Table II
MIXES

VI. Scheduler Evaluation

A. Mix Design and Evaluation Methodology

To evaluate the effectiveness of the proposal we designed a set of ten mixes, being the number of benchmarks in each mix twice as large the number of cores. Table II shows the mixes and the associated $MPKI_{L1}$ and $MPKI_{L2}$ as well. As analyzed above, there are benchmarks with poor memory requirements that have scarce misses in L1 and L2 caches. Thus, if the mix is built only using these benchmarks, there will not be any appreciable difference among the studied schedulers. As opposite, if all the benchmarks in the mix have a high MPKI in both L1 and L2 caches, then the scheduler will be forced to launch memory-hungry benchmarks together, leaving little room to improve performance. Therefore, a good mix should include a subset of memory-bounded benchmarks mingled with a subset of benchmarks with few memory requirements.

On the other hand, remark that the execution time widely varies among the different benchmarks. For example, some benchmarks like *wrf* or *sphinx* are completely executed in less than 1 second, while other benchmarks like *tonto* take more than 400 seconds. This fact must be addressed in the evaluation methodology, since a policy prioritizing the longest jobs would provide the best performance in most workloads. To deal with this evaluation shortcoming, we consider that each benchmark in the mix executes in the experiment as many instructions as it executes during a fixed amount of time when it is running alone. In particular, we gathered the number of instructions that each benchmark runs during two minutes. Then, during the mix scheduling experiment, if the execution of a given benchmark is shorter than two minutes it is relaunched by the scheduler. On the contrary, if it is longest, the benchmark is stopped and the remaining instructions are discarded. In addition, since benchmarks with very short execution time would be relaunched several tens of times, such benchmarks were discarded to build the mixes.

For evaluation purposes, we compared the performance of the cache hierarchy aware proposal against two schedulers: a memory-aware scheduler and the Linux OS scheduler. Notice that according to this evaluation methodology, the benchmarks must be killed and relaunched several times after they execute a given number of instructions. Therefore,

execution time cannot be directly measured using Linux scheduler, since we would need to stop the processes at the time they execute the target number of instructions. To solve this issue, we implemented an user-level pseudoscheduler that obtains native Linux turnaround times. This pseudoscheduler stops the processes and reads the number of executed instructions by each process. Then, the pseudoscheduler lets the process to follow its execution, kills the process or relaunches it depending on the number of executed instructions is less than the target, greater than the target or the process already finished its execution, respectively. Notice that these actions do not take scheduling decisions, which are taken by Linux scheduler. In this way, all the studied schedulers are fairly evaluated since all of them use the same quantum length, and stop and relaunch benchmarks in the same way. Quantum length was fixed to 200ms in the experiments.

B. Scheduler Performance

Figure 8 shows the speedup achieved by both the proposed scheduler and the memory-aware scheduler over the native Linux scheduler, considered as baseline. As observed, regardless the benchmark, the proposal always provides better performance than the memory-aware scheduler. The achieved speedup widely varies across the mixes, ranging from 1.8% to 6% and from 3.3% to 9%, for the memory-aware scheduler and the proposal, respectively. In short, the proposed algorithm, which considers the cache hierarchy contention points, performs better than the memory-aware scheduler. Furthermore, in some cases (e.g., mix 6 and mix 8), the proposal doubles the speedup of the memory-aware scheduler.

The main reason for this performance enhancement is that the cache-hierarchy aware policy balances transactions among the contention points across 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.

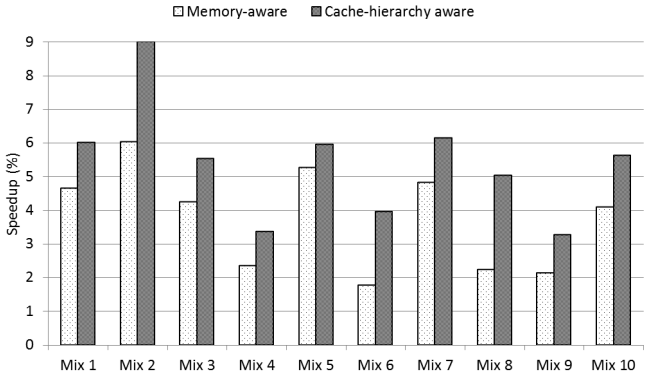


Figure 8. Speedup over native Linux OS

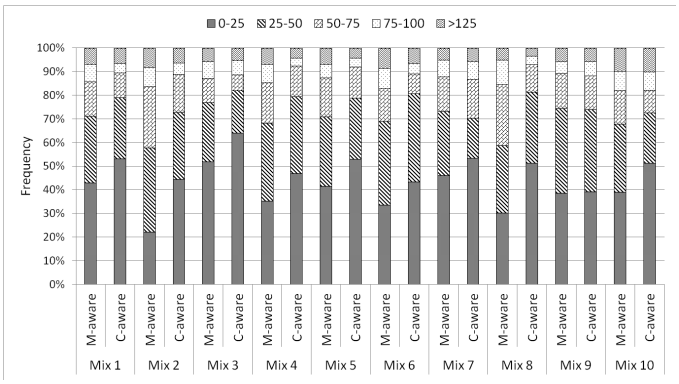


Figure 9. BTR differences between the L2 shared caches. Legend: M-aware (memory-aware), C-aware (cache hierarchy aware)

To estimate how good job balancing works, we measured the BTR arriving to both L2 caches (produced by L1 misses) and calculated their difference. Figure 9 presents the results. The histogram represents the frequency of the BTR_{L2} difference between both L2 caches for both schedulers across all the mixes. Results are presented in intervals of 25 transactions per μs. The higher the frequency of the lower intervals the better the transactions are balanced between L2 caches. In this way, the L2 bandwidth contention is reduced, which turns in performance enhancements.

For instance, if we the compare memory aware bar versus cache-hierarchy aware bar in mix 1, we can observe that with the memory-aware scheduler 40% of times (black bar) the BTR difference between both L2 caches was less than 25 (i.e., [0-25]) 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, with cache-hierarchy aware, differences lower than 25 transactions/us grows up to 50% of times, resulting in better BTR distribution and better performance.

As expected, results show a strong correlation between

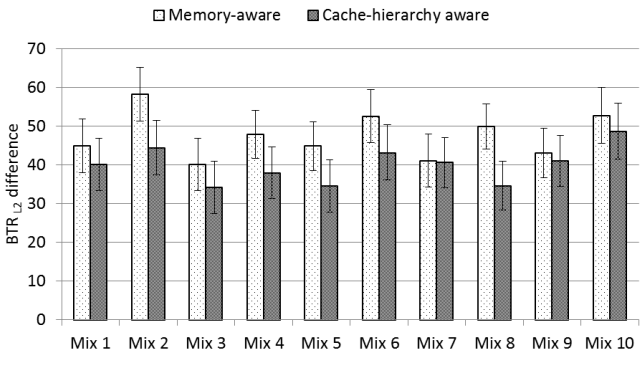


Figure 10. Average and variance of the difference between the BTRs of the L2 caches

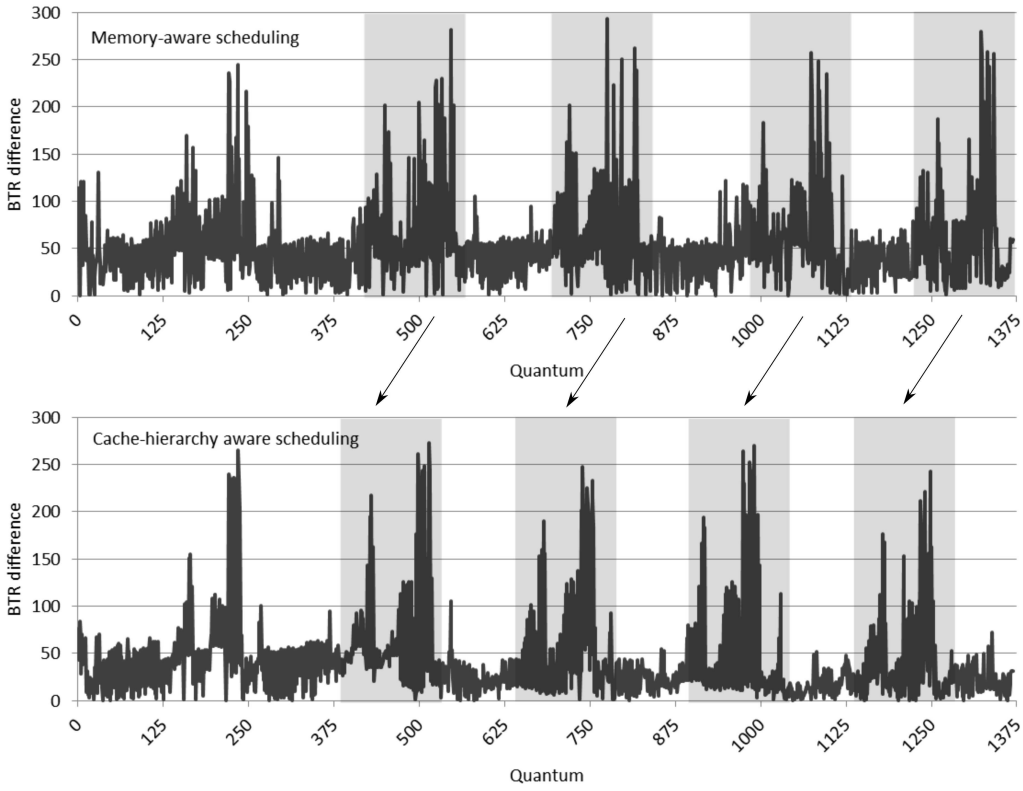


Figure 11. BTR_{L2} difference evolution with time

the frequency distribution and the speedup presented above. For instance, mix 2 and mix 8 present the widest distribution variations between both schedulers, which translates in the highest speedup variations. This can be clearly appreciated in the lowest interval (i.e., 0-25). The only exception is mix 9, which has similar frequencies for both schedulers. However, even in this case, the cache hierarchy aware scheduler still outperforms the memory-aware policy. This can be explained with the results shown in Figure 10, which presents the average and variance of the difference between the BTRs of both L2 caches and, as observed, the proposal presents a lower average.

To provide a sound understanding of BTR balancing, let's look inside the dynamic execution of a mix. In particular, let's focus on mix 2 where the proposal improves by 50% the speedup achieved by the memory-aware scheduler. Figure 11 shows the monitored results during the first 1375 quanta of execution. The plots, in the upper and lower sides of the Figure, show the results for the memory aware and cache hierarchy aware schedulers, respectively.

An interesting observation is that peak BTR differences, which are mainly caused by the *mcf* benchmark, appear before in the proposal (see grey boxes). Notice too that this progress in time is not achieved at expenses of increasing the peak heights, but the heights are reduced too. Looking at the lower plot, it can be appreciated that in many intervals the BTR difference falls always below 50 transactions per μs. Notice that this horizontal line (BTR difference = 50) cannot be seen in the upper plot. To provide insights in this analysis, Figure 12 zooms the first 160 quanta. It can be observed, for instance, that the longest interval below 50 is around 24 quanta in the proposal (from 120 to 144 aprox.), which is more than twice as large than that of the memory-aware scheduler in this execution. Finally, it can be also appreciated that peaks are both more frequent and higher in the memory-aware scheduler than in the cache-hierarchy aware scheduler.

VII. 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 characterized the benchmarks behavior on a commercial CMP processor, varying the location of the contention points across the memory hierarchy, as well as their intensity. We found that, contrary to expected, and depending on the co-runners, some workloads are more

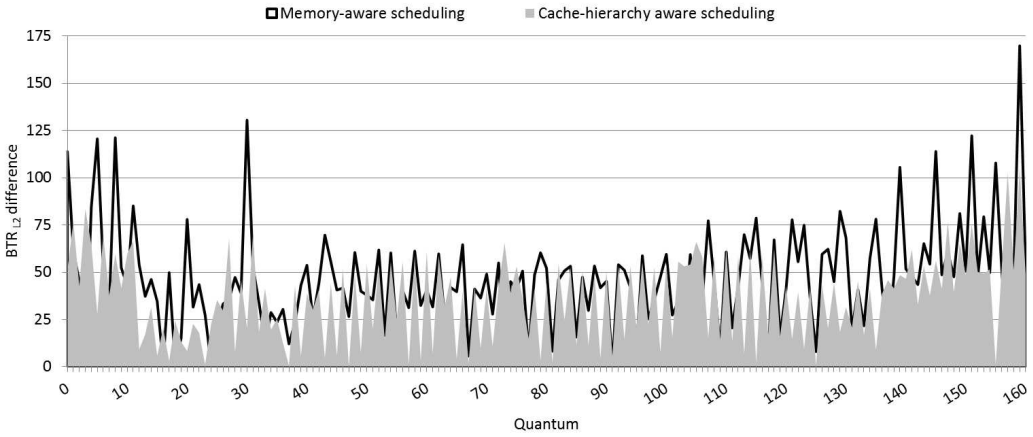


Figure 12. BTR_{L2} difference in the first 160 quanta

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, since the current industry trend is to increase the number of cores and their multithreading capabilities, as well as enlarging the size and latency of the shared caches. To deal with this performance problem, we have proposed a multi-level scheduling policy for a generic CMP that selects the jobs to be scheduled taking into account the memory bandwidth and balances bandwidth requirements across the cache hierarchy in order to reduce the potential contention points.

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 scheduling proposal does not require any additional hardware support, instead it only needs the information provided by some hardware counters already available in current microprocessors.

Experimental results show that, compared to the native Linux scheduler, the proposal achieves speedups ranging from 3.3% to 9%. Moreover, in some cases the proposal doubles the speedup achieved by a memory aware scheduler that does not take into account the cache hierarchy. Finally, remark that these results have been obtained in a commercial quad-core Intel Xeon X3320, whose bandwidth requirements for the cache hierarchy are expected to be much more lower than in future many-core multithreaded processors.

ACKNOWLEDGMENTS

This work was supported by the Spanish MICINN, Consolider Programme and Plan E funds, as well as European Commission FEDER funds, under Grants CSD2006-00046 and TIN2009-14475-C04-01.

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 *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, oct. 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 the 2004 IEEE/ACM International Conference on High Performance Computing (HiPC2004*, 2004, 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=1148882.1148884

[5] 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 *Solid-State Circuits Conference, 2007. ASSCC '07. IEEE Asian*, nov. 2007, pp. 14–17.

[6] 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.

[7] J. Shin, K. Tam, D. Huang, B. Petrick, H. Pham, C. Hwang, H. Li, A. Smith, T. Johnson, F. Schumacher, D. Greenhill, A. Leon, and A. Strong, "A 40nm 16-core 128-thread cmt sparc soc processor," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, feb. 2010, pp. 98 –99.

[8] E. Koukis and N. Koziris, "Memory and network bandwidth aware scheduling of multiprogrammed workloads on clusters of smps," in *Proceedings of the 12th International Conference on Parallel and Distributed Systems - Volume 1*, ser. ICPADS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 345–354. [Online]. Available: http://dx.doi.org/10.1109/ICPADS.2006.59

[9] F. Pinel, J. E. Pecero, P. Bouvry, and S. U. Khan, "Memory-aware green scheduling on multi-core processors," in *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ser. ICPPW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 485–488. [Online]. Available: http://dx.doi.org/10.1109/ICPPW.2010.71

[10] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 423–432. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2006.49

[11] G. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, feb. 2002, pp. 117 – 128.

[12] J. Chang and G. S. Sohi, "Cooperative cache partitioning for chip multiprocessors," in *Proceedings of the 21st annual international conference on Supercomputing*, ser. ICS '07. New York, NY, USA: ACM, 2007, pp. 242–252. [Online]. Available: http://doi.acm.org/10.1145/1274971.1275005

[13] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 111–122. [Online]. Available: http://dx.doi.org/10.1109/PACT.2004.15

[14] M. Sato, I. Kotera, R. Egawa, H. Takizawa, and H. Kobayashi, "A cache-aware thread scheduling policy for multi-core processors," in *Parallel and Distributed Computing and Networks (PDCN), 2009 IASTED 8th International Conference on*, feb. 2009, 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, February 2010. [Online]. Available: http://doi.acm.org/10.1145/1646353.1646371

[16] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ser. ASPLOS '10. New York, NY, USA: ACM, 2010, pp. 129–142. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736036

[17] 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 *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, jan. 2010, pp. 1 –11.

[18] P. Padala, "Playing with ptrace," *Linux Journal*, 103, 2002.

[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. [Online]. Available: http://stacks.iop.org/1742-6596/119/i=4/a=042017