FlexDCP: a QoS framework for CMP architectures

Miquel Moreto
Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
mmoreto@ac.upc.edu

Francisco J. Cazorla
Barcelona Supercomputing Center (BSC), Barcelona, Spain
francisco.cazorla@bsc.es

Alex Ramirez
UPC, BSC, Barcelona, Spain
aramirez@ac.upc.edu

Rizos Sakellariou
University of Manchester, United Kingdom
rizos @cs.man.ac.uk

Mateo Valero
UPC, BSC, Barcelona, Spain
mateo@ac.upc.edu

Abstract

Current multicore architectures offer high throughput by increasing hardware resource utilization. As the number of cores in a multicore system increases, providing Quality of Service (QoS) to applications in addition to throughput is becoming an important problem.

In this work, we present FlexDCP, a framework that allows the Operating System (OS) to guarantee a QoS for each application running in a chip multiprocessor. FlexDCP directly estimates the performance of applications for different cache configurations instead of using indirect measures of performance like the number of misses. This information allows the OS to convert QoS requirements into resource assignments. Consequently, it offers more flexibility to the OS as it can optimize different QoS metrics like per-application performance or global performance metrics such as fairness, weighted speed up or throughput.

Our results show that FlexDCP is able to force applications in a workload to run at a certain percentage of their maximum performance in 94% of the cases considered, being on average 1.48% under the objective for remaining cases. When optimizing a global QoS metric like fairness, FlexDCP consistently outperforms traditional eviction policies like LRU, pseudo LRU and previous dynamic cache partitioning proposals for two-, four- and eight-core configurations. In an eight-core architecture FlexDCP obtains a fairness improvement of 10.1% over *Fair*, the best policy in the literature optimizing fairness.

Categories and Subject Descriptors C.1 [*Computer Systems Organization*]: Processor Architectures; D.4 [*Software*]: Operating Systems; B.3.2 [*Memory Structures*]: Design Styles—cache memories

General Terms Design, Performance, Measurement

Keywords Multicore Systems, Cache Partitioning, Quality of Service, Performance Predictability, Operating Systems

1. Introduction

The current collaboration between the Operating System (OS) and multithreaded architectures is inherited from the traditional collaboration between the OS and multiprocessors: The OS perceives the different cores in a chip multiprocessor (CMP) (7) and the hardware contexts in a simultaneous multithreading architecture (SMT) (28; 33) as multiple, independent virtual processors. Thus, the OS is not aware of the resource sharing problem and schedules threads onto what it regards as independent processing units. However, in multithreaded architectures the number of instructions executed by a thread depends on the activity of the co-scheduled threads. If no explicit control over shared resources is exercised, the performance of applications becomes unpredictable. Several studies (4; 23) show that in both SMTs and CMPs the performance of a task heavily depends on the workload[1] it is executed in. To deal with this performance variability problem, the OS should be able to exercise more control over how threads share the internal resources of the processor. More interaction is needed between the architecture and the OS to allow the latter to provide some *Quality of Service* (QoS) to applications (25).

General-purpose computing is moving off of desktops onto diverse devices such as cell phones, digital entertainment centers, and data center servers. The interaction between the OS and the architecture must be flexible enough to cover different scenarios where the concept of QoS has different meanings. For instance, in a high throughput server scenario the target to maximize is system performance or *overall QoS* (26; 15; 27) that can be measured with metrics like fairness, weighted speed up or throughput. In other scenarios like multimedia and real-time systems, per-application or *individual QoS* is required (24; 12; 6). Finally, there are intermediate situations like soft real-time systems with *hybrid QoS* requirements, where some applications need an individual QoS and the remaining ones need a global QoS (6). Hence, providing QoS to a wide range of scenarios is an important challenge for future multicore architectures.

Recently, some authors have proposed an abstract, generic framework for future many-core architectures that allows the OS to explicitly manage resource allocation (25). This framework incorporates features from previous QoS frameworks and provides a general approach to build new interfaces between the OS and the architecture. Figure 1 shows the main components of this framework.

[1] A workload is a set of processes running simultaneously on the CMP.

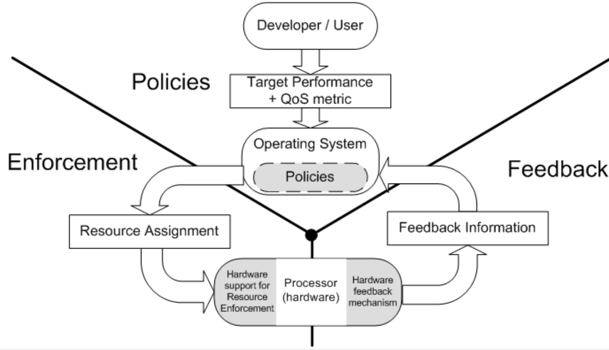


Figure 1. Generic framework to manage shared resources in a CMP architecture.

1. **Policies**: they are implemented primarily in software. Policies should translate application and system objectives into resource assignments, thereby managing system resources.
2. **Enforcement mechanisms**: they securely multiplex, arbitrate, or otherwise distribute hardware resources in order to satisfy the resource assignments. Their main task is to enforce that each thread receives the amount of resources established by the policies.
3. **Feedback mechanisms**: they inform the software about the global resource usage, which can be used by the OS to find a new resource assignment to accomplish with the QoS requirements of the running applications.

A common characteristic of all previous OS/architecture interface proposals (6; 12; 24) is that they have not addressed one of the key points of this approach: *converting a given QoS target by applications into a resource assignment* (25). For example, let us assume that a given task has to be executed before a deadline *d*. None of the proposed interfaces provides to the user a method to convert this high-level QoS requirements into a resource assignment for this task so that it meets the deadline *d*. The OS/architecture interfaces proposed so far assume that either the applications will be able to specify a target usage of each shared resource or the OS will know somehow the way to convert performance targets into resource assignments. The former situation is not possible in most cases, as applications are normally architecture-independent. Hence, the developer cannot provide the exact amount of resources that an application will require to obtain a target performance. In the latter situation, the OS job scheduling has to be architecture independent to be portable between different architectures. With current OS/architecture interface proposals, the user establishes a given initial resource partition for the task. At the end of each time quantum, the OS checks whether the task can accomplish its QoS objective with this resource partition, increasing the amount of resources given to it if this is not the case. This iterative process can take long and, as applications may change their behavior, this process has to be repeated frequently. As a consequence, applications are constantly executed in a sub-optimal resource partition.

In this paper, we propose *FlexDCP*, a flexible framework that represents the first implementation of a complete QoS framework. On the one hand, we propose an effective feedback mechanism that allows translating QoS requirements from the user into a hardware resource assignment in a single step. FlexDCP is the first framework to do so. On the other hand, FlexDCP supports all kind of QoS requirements in CMP architectures with a shared cache. FlexDCP can optimize any target metric related to IPC, leading to the best performance results for at least three different targets (ensuring an individual QoS level, fairness and throughput). This flexibility is not possible with previous proposals, which focus on improving a particular metric or cannot ensure a target individual QoS level.

In the FlexDCP framework, the architecture provides the OS with the performance of running applications under the current cache assignment, as current performance counters do. In addition, FlexDCP uses additional hardware that also provides the OS with the performance that the running applications would have with all other possible cache size assignments. By reading this information, the OS can compute the performance degradation or improvement of each application when moving to another cache configuration. This allows the OS to translate QoS requirements into resource assignment, without profiling the application or forcing the OS to know the internal details of the architecture, making it totally architecture independent. Nor do application developers require specifying the exact amount of resources that their applications must use, making our solution closer to more realistic scenarios.

The main contributions of this work are the following:

1) **Flexibility**: We propose a new feedback mechanism that predicts the performance of running applications under cache partitions different from the current one. FlexDCP can maximize overall QoS metrics like harmonic mean of relative IPCs [2], weighted speed up or throughput, or ensure an individual QoS metric. Previous proposals do not offer this flexibility.

- *Individual QoS*: In contrast to previous work, FlexDCP allows jobs to run at a certain percentage of their maximum speed, regardless of the workload in which these jobs are executed. Our results, on a CMP scenario with a shared L2 cache, show that FlexDCP successfully accomplishes with the target IPC in 94% of the cases considered, reaching an IPC that is 1.48% lower than the objective IPC in the remaining 6% of the cases.

- *Global QoS/Scalability*: Our results show that previous proposals based on indirect metrics of performance provide diminishing returns as the number of cores sharing the L2 cache increases. FlexDCP obtains sustained throughput and fairness improvements over LRU and previous proposals on the two-, four- and eight-core architecture setups used in this paper. In the eight-core architecture FlexDCP obtains a fairness improvement of 10.1% over *Fair*, the best policy in the literature optimizing fairnes. When optimizing throughput, FlexDCP obtains improvements of 11% on average over MinMisses, the best policy in the literature improving throughput.

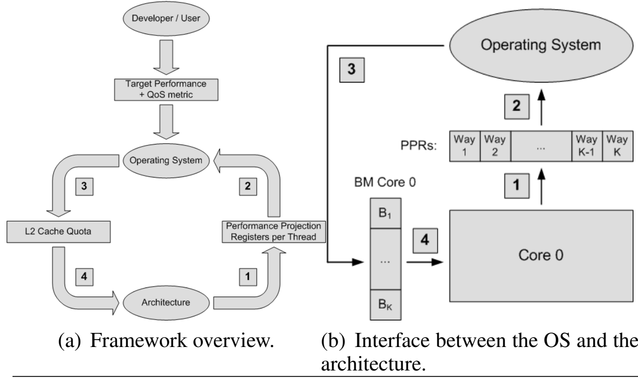
2) **Granularity Analysis**: In this paper we show that the time granularity at which the resource assignment decisions are taken has a significant impact on performance. Wrong decisions are very costly, mainly when the time granularity is high. Meanwhile making resource assignment decisions too frequently also affects overall performance. In this paper, we give a complete analysis on how to tune this decision period in order to obtain the highest performance.

The rest of this paper is structured as follows. Section 2 presents our new framework that ensures both individual and global QoS. Section 3 describes the experimental environment while in Section 4 simulation results are discussed. Section 5 introduces the related work. Finally, Section 6 summarizes our results.

2. FlexDCP QoS Framework

FlexDCP is a framework that allows the OS to guarantee a QoS for each application in a CMP architecture. FlexDCP provides the OS with the necessary information to convert user's QoS requirements into resource allocation. In particular, FlexDCP focuses on

[2] The relative IPC of a thread is the ratio of its IPC when it runs in a workload with respect to its IPC when it runs in isolation using all resources.



(a) Framework overview. (b) Interface between the OS and the architecture.

Figure 2. FlexDCP: a QoS framework for CMP architectures.

the shared caches as one of the main sources of interaction between threads in CMP architectures. The architecture provides the OS with the performance of running applications under the current L2 cache assignment, as current performance counters do, and the performance that the running applications would have with all other possible cache size assignments. With this information the OS can compute the performance degradation or improvement of each application when moving from the current cache allocation, *currentCA*, to a new cache allocation, *newCA*, simply by computing: $execution_time_reduction = \frac{IPC_{currentCA}}{IPC_{newCA}}$. Given that the IPC values are provided by the architecture and the OS works with the ratio between them, the OS does not need to know the internal details of the architecture, making it architecture independent.

Figure 2(a) describes our QoS framework. First, developers or users determine the target performance of the application and the QoS metric to optimize. In some scenarios, like multimedia or real time applications, different instances of the same application have approximately the same performance (11). Thus, the user knows beforehand the full speed of the application and can provide it to our framework. The individual target performance determines the minimum performance that the application requires, while the QoS metric determines what to do with unassigned resources. If no individual target performance is specified, the QoS metric will guide resource assignment. For instance, fairness or throughput might be optimized.

Next, the OS schedules applications according to their QoS necessities. In our experiments we assume that the workload has already been chosen by the OS. When different applications start executing in the CMP architecture, an initial partition of the shared L2 cache is decided. If the OS has no prior knowledge of the applications, resources are evenly partitioned among threads. Thus, it assigns $\frac{Associativity}{Number\ of\ Cores}$ ways of the shared L2 cache to each thread. If the OS already has prior knowledge of the applications, it can decide the initial partition based on that information.

Next, dedicated hardware estimates the performance of the application running in each core with all other possible cache allocations. We denote these performance estimations *Performance Projections*. This hardware mechanism is detailed in Section 2.2. Each thread stores its performance projections in a set of registers visible to the OS. We call these registers *Performance Projection Registers* (PPR). For a K-way associative cache, there are K 64-bit PPRs (Figure 2(b), Step 1).

After this estimation period, the OS analyzes the values of the PPRs of each core (Figure 2(b), Step 2). Using this information, the OS decides a new partition for the next period (Figure 2(b), Step 3). We assume that performance in the current measuring period is representative of the performance of the next period. Thus, the optimal partition for the last period will be chosen for the following period. FlexDCP assigns to each thread the required cache quota to satisfy its individual QoS requirements[3]. Then, the remaining resources are assigned among all threads according to the overall QoS metric. Throughout our work, we determine the optimal partition for our proposal as well as for previous work. We take into account this time overhead when reporting performance results.

Table 1. Variability in the impact on performance of L2 misses.

Benchmark		Ways	Misses	IPC
crafty	config. 1	10	31K	1.689
	config. 2	14	21K	1.707
	variation	+4	−32%	+1.1%
facerec	config. 1	7	2M	0.924
	config. 2	16	1.2M	1.16
	variation	+9	−40%	+25.5%
equake	config. 1	1	10M	0.245
	config. 2	4	6M	0.266
	variation	+3	−40%	+8.6%
vpr	config. 1	15	714K	0.88
	config. 2	16	600K	0.966
	variation	+1	−16%	+9.7%

Finally, cache partitions are implemented at a way granularity with *column caching* (5), which uses a mask that marks the cache ways (or columns) reserved for each thread. When a thread experiences an L2 miss, the evicted line is the LRU line among the lines owned by that thread. When the OS decides the cache quota per thread, it writes the corresponding bit masks (BM) (Figure 2(b), Step 3). Each BM contains a bit per cache way and there is a BM per thread. If the k-th bit of the mask of a thread is set, the thread owns that way. Running threads can read from all cache lines and, consequently, correctness is ensured when updating bit masks (5). Other authors have used more flexible implementations like *Augmented LRU* (32). However, its hardware cost is considerable, as a counter per thread and set is needed. Thus, in this work we use column caching.

In the following section, we motivate the use of direct estimations of performance as the adequate metric to decide L2 cache partitions. Next, we give the hardware mechanism needed to obtain these performance projections. Finally, in Section 2.3 we discuss the adequate granularity of cache quota decisions.

2.1 Direct Vs Indirect Performance Metrics

A common characteristic of previous proposals is that they decide new cache partitions using indirect indicators of performance, mainly the number of L2 misses (5; 26; 32; 15; 29). However, the effect of L2 misses on performance varies depending on the application and even on the particular phase of the application.

To illustrate this idea, Table 1 shows the variation in performance and the number of misses for some benchmarks from the SPEC CPU 2000 suite as we vary the number of active ways, w. For this experiment, we simulate a single threaded architecture with a 16-way associativity 1MB L2 cache (see Section 3 for more details). The remaining $16 - w$ ways are simply switched off. For example, observe that when we move from 7 to 16 active ways, `facerec` reduces its number of L2 misses by 40%. Analogously, `equake` reduces misses by 40% when it moves from 1 to 4 ways. However, the effect on performance is different: the IPC of `facerec` increases by 25% while the performance of `equake` only increases by 9%. In contrast, we observe similar variation in performance for `vpr` and `equake` and the reduction in misses is different (16% and 40%). This different impact on performance also happens in case of `crafty`.

[3] We ensure at least one reserved way in the L2 cache for each application.

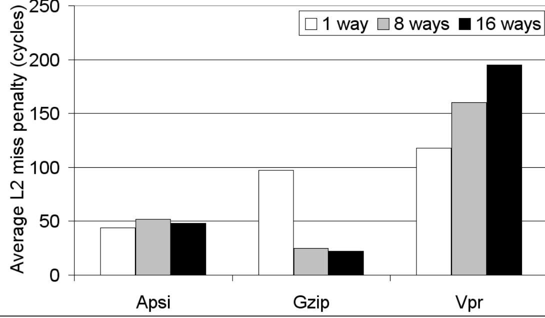


Figure 3. Average L2 miss penalty for apsi, gzip and vpr with three different L2 cache configurations.

Translating IPC into resource assignment has been identified as a challenging problem (6; 25). It has also been shown to be a key element in future multicore systems to improve the interaction between the OS and the architecture (25). A solution to this problem consists in using the average miss penalty of L2 misses in the current L2 cache configuration and assume it is constant for other configurations (35). Figure 3 shows the average miss penalty of L2 data misses when we vary the number of active ways of the L2 cache from 1 to 16 for three different benchmarks. This miss penalty significantly varies among L2 cache configurations because the clustering level of the L2 misses changes for different cache sizes: an isolated L2 miss has approximately the same miss penalty as a cluster of L2 misses, if they all fit in the reorder buffer (ROB) and thus can be served in parallel (14).

Instead, we propose to estimate this miss penalty at runtime using analytic models for superscalar processor performance. This mechanism is based on OPACU methodology (21) and allows predicting IPC at runtime for different L2 cache configurations without running all these configurations.

2.2 OPACU Methodology

OPACU methodology (Online Prediction of Applications Cache Utility) has been proposed to predict the IPC of an application in single threaded architectures for different cache configurations at runtime (21). In this paper, we use a CMP architecture where each core has private L1 data and instruction caches and share a dynamically partitioned L2 cache with uniform access time. We have adapted OPACU methodology to predict IPC values for each benchmark in a workload.

OPACU interprets the performance of an application as a Cycle Per Instruction (CPI) stack, composed of an ideal CPI (when no misses occur) and the CPI penalties for each type of hazard, including branch mispredictions, instruction cache misses and data cache misses (14). Here, as the L2 cache is partitioned, interactions between different threads are limited to the shared bus to access the L2 cache and main memory. The size of the L2 cache assigned to the thread varies, but the rest of the architecture remains constant. Thus, the ideal CPI is assumed to be independent of the cache configuration (it only depends on data dependencies of the particular application). It can also be assumed that the branch miss penalty of a particular thread remains constant for different cache sizes.

Thus, OPACU only considers the part of the model that concerns the cache hierarchy. The model considers L2 instruction and data misses separately: instruction misses are always serialized while data misses can be served in parallel if they fit in the ROB. Thus, the instruction miss penalty is constant (ΔD cycles, in absence of bus contention), while the average data miss penalty ($aDMP$) can be computed as $aDMP = \frac{N_D}{M_D} \cdot \Delta D$, where N_D is the total number of clusters of L2 data misses and M_D is the total number of L2 data misses.

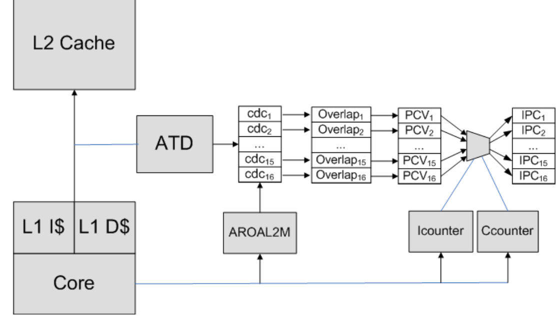


Figure 4. OPACU's hardware implementation for one core with a 16-way L2 cache.

OPACU uses a sampled Auxiliary Tag Directory (ATD) to obtain the number of misses per L2 configuration as in (32; 26; 15; 24) and a reduced number of hardware counters to determine if L2 data misses are clustered or not. Three counters per core are needed: number of instructions, cycles and average ROB usage after an L2 miss (*AROAL2M*). Three counters per cache assignment are also needed: last L2 miss identifier (cdc_i), number of clusters ($overlap_i$) and total waiting cycles due to an L2 miss (PCV_i). When a load accesses the L2, the sampled ATD is used to determine if it would be a miss in other possible cache assignments. Using the last L2 miss identifier, it can be determined whether a new cluster of misses begins or not. The number of clusters and lost cycles is updated accordingly. Using this information, IPC predictions are obtained and stored in the PPRs. Figure 4 illustrates this mechanism.

For a four-core CMP with a shared 1MB 16-way L2 cache, OPACU needs less than 1KB of total storage per core (including a sampled ATD (26) and all the required hardware counters (21)). The most expensive piece of hardware of this mechanism is the sampled ATD. Given its wide use, significant efforts in the community have been recently devoted to reduce the sampled ATD's area to tens of bytes per thread (26; 12; 24; 6). Some authors have embedded the ATD inside the L2 cache, devoting some sets to monitor each thread (13). Using this approach, the hardware cost of OPACU would be reduced to 204 bytes per core with a 16-way L2 cache. Moreover, in our view, energy consumption, rather than area, is a main problem in future processor's design. Only 1 entry in the sampled ATD is active at a time, thus its energy consumption is low. As a consequence, implementing sampled ATDs in future processors will be feasible.

The main limitation of this analytic model is that it does not take into account dependencies between different data misses, as happens in pointer chasing. Despite of this limitation, OPACU obtains high accuracy for different cache configurations, with an average relative error of 3.11% when using the whole SPEC CPU 2000 benchmark suite. `Parser` and `twolf` are the benchmarks with highest errors (9.8% and 16%) due to the simplifications of the analytic model. The prediction error for a fixed number of assigned ways is 3.11% and presents higher errors when just one or two ways are being used (8% and 5% respectively) (21).

Using OPACU methodology to estimate performance is not a limitation of our framework as other models can be developed to obtain more accurate performance estimations. However, this work shows that using performance projections to decide cache partitions is more adequate and leads to better performance than previous proposals guided by miss rates.

2.3 Granularity of Cache Quota Decisions

The frequency of cache partitioning decisions directly impacts the performance improvement obtained by the mechanism. In this section we show three possible implementations depending on the de-

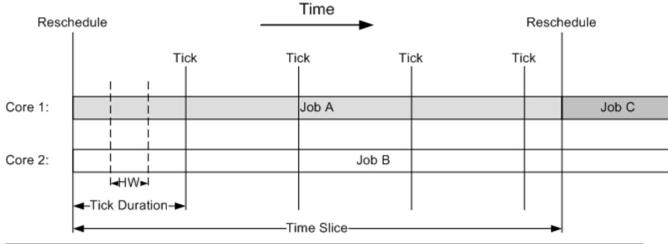


Figure 5. Partitioning granularities in a two-core architecture.

Algorithm 2.1: TIMER INTERRUPT()

1- Save architecture state.
2- Call scheduler_ticks()

2.1- Update counters and statistics.
2.2- Check if there are ready threads with higher priority.
2.3- Decrement quantum of time and check if the quantum has expired.
2.4- Balance load between different task queues.

*3- **Invoke microcode to decide L2 cache partition**.*
4- Restore architecture state.

sired granularity of decisions. Figure 5 shows these three possible alternatives.

1) Hardware granularity. The decision logic used to decide new partitions can be implemented in hardware (12). With this solution, the OS specifies the desired target performance at the time slice boundary and the hardware decides new L2 cache partitions at a smaller time granularity. Consequently, this solution provides a quick response time to phase changes. If there is no time overhead in deciding new partitions, the hardware solution is the best one. However, as the number of cores and L2 associativity increase, the time overhead of making a new decision also increases. As a consequence, deciding new partitions too frequently can negate the performance benefits. Furthermore, implementing a flexible decision algorithm with different metrics to optimize with a complexity-effective hardware is difficult.

2) Interrupt granularity. The second option consists in programming a periodic interrupt to decide new L2 cache partitions (27). With this option the frequency of partition decisions can be chosen, but we have to pay the overhead of interrupting the application and executing the interrupt handler. Thus, we have lower hardware complexity than the hardware solution and a higher time overhead as decisions are made in software. With this solution, the time overhead can become a problem as interrupt handlers must not take long: While the interrupt is running, other interrupts are inhibited and might be lost, which can be a critical problem. Instead of adding the decision algorithm inside the interruption handler, we propose to use a microcode piece of code that is in charge of deciding the new partition. With this solution other interrupts are not lost. The idea is similar to *millicode* (8) or co-designed virtual machines (31), which have a *concealed memory* reserved in main memory at boot time and the conventional applications are never informed of its existence. The OS can program this concealed memory, which gives more flexibility to the framework. We modify the timer interrupt to invoke this microcode when an OS tick occurs. The code that resides in the concealed memory can take control of the hardware and decide new L2 cache partitions. Algorithm 2.1 shows the different actions done when an OS tick occurs (3).

Table 2. Performance improvement over LRU in a 4-core CMP with a time overhead of 5000 cycles.

Granularity	Decision period	Over head	Max perf. improvmnt.	Real perf. improvmnt.
Hardware	100K	5%	10.95%	5.67%
	500K	1%	10.65%	9.54%
Interrupt	1M	0.5%	10.53%	9.98%
	5M	0.1%	10.15%	10.04%
Scheduler	50M	0.01%	8.51%	8.499%
	100M	0.005%	8.00%	7.99%

3) OS quantum granularity. Finally, L2 cache partitions can be decided at time slice boundaries (32). In this case, the OS already interrupts the application, reducing the time overhead of the solution. However, the frequency of decisions might be too coarse to adapt to phase changes. In Linux 2.6, the timer interrupt can be configured to different periods: 1ms, 3.33ms, 4ms (default) and 10ms. The time slice duration depends on the thread priority and is between 5ms and 800ms (100ms by default) (3). At a frequency of 2GHz, this corresponds to a range from 2 to 20 million cycles for the timer interrupt (8 million by default), and a range from 10 to 1600 million cycles for the time slice (200 million by default). If deciding cache partitions in every OS tick becomes too expensive, decisions can be made when the OS schedules threads. Thus, we have to make performance projections visible to the scheduler. We propose to store these projections in the *task struct*, which stores information of all processes alive in the system. With these performance projections, the scheduler can restore the values of the performance registers in a previous time slice. This idea prevents deciding wrong L2 cache partitions when a new thread is scheduled. Finally, the microcode is invoked to decide the new cache partition.

The optimal granularity at which a policy decides new cache quotas depends on the application under consideration and also on the time overhead of making a new decision. Instead of deciding new cache partitions evaluating all possible combinations and choosing the one that optimizes a given metric (exhaustive search), we implement an algorithm that uses dynamic programming techniques to reduce the time overhead of deciding new partitions [7]. Finding out the optimal partition for just two cores is straightforward as we have to check just K partitions. Thus, we can compute the optimal number of misses when w ways are assigned to these two threads, $misses(w)$, in w steps. This function is independent of the other threads, allowing us to build this curve in parallel with the other thread pairings. Next, the same algorithm is repeated for the new histograms of misses. For each pairing, the complexity of the algorithm is $O(K^2/2)$, which is repeated $\frac{N}{2} + \frac{N}{4} + \ldots + 1 = N - 1$ times.

We have measured using the PIN instrumentation framework (17) the number of dynamic instructions executed by this algorithm when no individual QoS is specified (worst-case scenario). We have evaluated different number of cores (2, 4 and 8) and associativities (16, 24 and 32). With a 16-way cache, the number of executed instructions is less than 5000, while for a 32-way cache this number increases up to less than 20000 instructions. This algorithm only reads the PPRs and decides L2 cache quota. Hence, no data cache miss is caused. The number of static instructions is between 50 and 100, causing little pollution in the instruction cache. In the case of current CMP architectures with up to 8 cores, the time overhead of this algorithm is less than 5000 cycles (assuming an IPC of 1 instruction per cycle).

Next, Table 2 reports the performance improvements over LRU in a four-core architecture with a 1MB 16-way L2 cache when optimizing for throughput. The CMP configuration and workloads are detailed in Section 3. First, we give the ideal performance improvement when no time overhead is considered to make new cache decisions (fourth column in Table 2). Next, we evaluate two possible

decision periods corresponding to the three possible implementations of our framework. We consider 100 and 500 thousand cycles for the hardware solution, 1 and 5 million cycles for the interrupt solution, and 50 and 100 million cycles when deciding partitions at the scheduler granularity.

On the one hand, wrong decisions are very costly, mainly with high granularities. On the other hand, making decisions too frequently also affects overall performance. On average, using the interrupt solution provides the highest improvement. For that reason, we propose to use this granularity and decide cache partitions every 5 million cycles. Even with a configuration with 8 cores and a shared 32-way associative cache the overhead is under 0.4% with this granularity. In this paper, we consider the time overhead of deciding new cache partitions when reporting results.

2.4 Scalability of FlexDCP

Currently there are processors with 32- and 64-way associative caches: Niagara T2 (32-way (2)), AMD Barcelona (32-way L3) or ARM920T (64-way (1)). Furthermore, in future manycore architectures, we do not expect to have dozens of cores directly sharing the same cache due to limitation on bandwidth and capacity of the cache. The cores will be clustered in reduced groups of cores sharing a cache. Hence, we believe that our mechanism will work well with manycore architectures.

As the number of cores increases, the time overhead of evaluating all possible partitions also increases. Previous work has proposed several heuristics to determine partition candidates with low time overhead and without losing performance (32; 26). In this paper we have implemented a dynamic programming algorithm that drastically reduces the time overhead of evaluating all possible partitions. The time overhead is now comparable to previous proposed heuristics. Further reducing this overhead is part of our future work.

3. Experimental Environment

3.1 Simulation Configuration

In this study we focus on a CMP with two, four and eight cores. We use a two-level cache hierarchy, in which each core has a private data and instruction L1 caches and the unified L2 cache is shared among cores (15; 26; 32). Each core is single threaded and can fetch up to 8 instructions each cycle. It has 6 integer (I), 3 floating point (FP), and 4 load/store functional units and 32-entry I, load/store, and FP instruction queues. Each thread has its own 256-entry ROB and 256 physical registers. The processor frequency is 4 GHz. The first level data and instruction caches have 64B lines, 16KB size and 4 ways. For the L2 cache we used two configurations. A 1MB, 16-way associativity L2 cache. We also use a 2MB 32-way L2 cache as future CMP architectures will continue scaling L2 size and associativity.

The latency from L1 to L2 is 15 cycles, and from L2 to memory 300 cycles. We use a 32B width bus to access L2 and a multibanked L2 of 16 banks with 3 cycles of access time. We model a single channel to access main memory. Bus and banks conflicts are modeled.

We extended the SMTSim simulator (33) to make it CMP. We collected traces of the most representative 300 million instruction segment of each program, following the SimPoint methodology (30). We use the FAME simulation methodology (34) with a Maximum Allowable IPC Variance of 5%. This evaluation methodology measures the performance of multithreaded processors by reexecuting all threads in a multithreaded workload until all of them are fairly represented in the final IPC measured from the workload.

Table 3. Workloads belonging to each case for a 16-way 1MB and a 32-way 2MB shared L2 caches.

	1MB 16-way		
# Cores	Case 1	Case 2	Case 3
2	155 (48%)	135 (41%)	35 (11%)
4	624 (4%)	12785 (86%)	1541 (10%)
6	306 (0.1%)	219790 (95%)	10134 (4.5%)
8	19 (0%)	1538538 (98%)	23718 (2%)
	2MB 32-way		
# Cores	Case 1	Case 2	Case 3
2	159 (49%)	146 (45%)	20 (6.2%)
4	286 (1.9%)	12914 (86%)	1750 (12%)
6	57 (0.02%)	212384 (92%)	17789 (7.7%)
8	1 (0%)	1496215 (96%)	66059 (4.2%)

3.2 Workload Classification

We use the following two heuristics to classify workloads with 2, 4 and 8 benchmarks in three different groups (20).

Heuristic 1. $w_{P\%}(B)$ measures the number of ways needed by a benchmark *B* to obtain at least a given percentage *P*% of its maximum IPC (when it uses all L2 ways).

Heuristic 2. The $w_{LRU}(th_i)$ heuristic measures the number of ways given by LRU to each thread th_i in a workload composed of N threads.

From these heuristics three groups of workloads can be created:

Case 1. When $w_{90\%}(th_i) \leq w_{LRU}(th_i)$ for all threads. In this situation LRU attains 90% of each benchmark performance. Thus, it is intuitive that in this situation there is very little room for improvement.

Case 2. When there exist two threads A and B such that $w_{90\%}(th_A) > w_{LRU}(th_A)$ and $w_{90\%}(th_B) < w_{LRU}(th_B)$. In this situation, LRU is harming the performance of thread A, because it gives more ways than necessary to thread B.

Case 3. Finally, when $w_{90\%}(th_i) > w_{LRU}(th_i)$ for all threads, the L2 cache configuration is not big enough to ensure that all benchmarks will have at least 90% of their peak performance.

From the SPEC CPU 2000 suite we generated all possible 2, 4, 6 and 8 workloads without repeating benchmarks in each workload. In Table 3 we show the total number of workloads that belong to each case for the different configurations with a 1MB 16-way and a 2MB 32-way L2 cache. Note that with different L2 cache configurations, the value of $w_{90\%}$ will change for each benchmark. An important conclusion from Table 3 is that as we increase the number of cores, there are more combinations that belong to the second case, which is the one with more improvement possibilities.

To evaluate our proposals, we use four different configurations. We denote these configurations *2C* (2 cores and 1MB 16-way L2), *4C-1* (4 cores and 1MB 16-way L2), *4C-2* (4 cores and 2MB 32-way L2) and *8C-2* (8 cores and 2MB 32-way L2).

3.3 Performance Metrics

Depending on the scenario, the most adequate metric to measure QoS may be different. For instance, in a high throughput server scenario, we want to be fair among all threads. As performance metric for this scenario, we use the harmonic mean of relative IPCs to measure fairness, which we denote *Hmean*. We use *Hmean* instead of weighted speed up because it has been shown to provide better fairness-throughput balance than weighted speed up (18). We also use the sum of individual IPCs, known as throughput, as a complementary metric. In other scenarios like real-time systems, we need to offer an *individual QoS* for each application. We use the relative IPC to measure individual QoS, which is computed as $\frac{IPC_{CMP}}{IPC_{alone}}$. Finally, there are intermediate situations (soft real-time systems), where some applications need an individual QoS and the remaining ones need a global QoS.

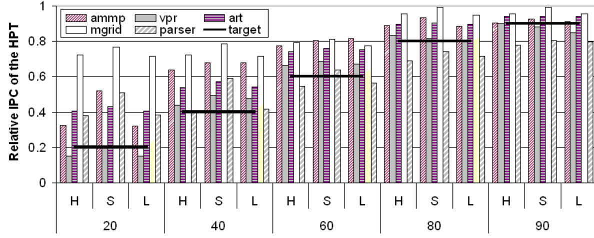


Figure 6. Predictable Performance in the *4C-2* configuration.

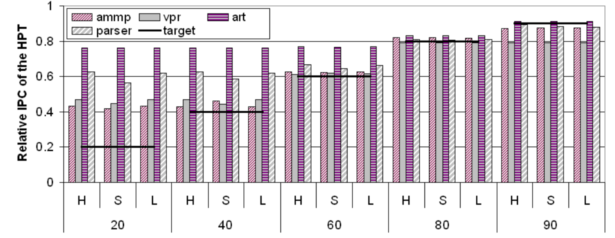


Figure 7. Predictable Performance in the *4C-1* configuration when the target IPC is specified beforehand.

4. Evaluation Results

4.1 Ensuring an Individual Quality of Service

Thanks to the flexibility of the FlexDCP framework, we are able to control the performance of an individual application when executed with other applications. Our framework allows the OS to run jobs at a certain percentage of their maximum speed, regardless of the workload in which these jobs are executed and without dedicating all shared resources to them. Thus, non-time-critical jobs can make significant progress as well and without significantly compromising overall performance.

To evaluate the individual QoS results of our proposal, we generate workloads containing four SPEC CPU 2000 benchmarks. One of these benchmarks is a *High Priority Thread* (HPT). Our objective is to force the HPT to run at a given *target percentage* of its *full speed* (IPC when it runs alone in the architecture). This full speed is estimated with the runtime mechanism to predict performance, which cannot be estimated in previous proposals. The HPT runs together with other Low Priority Threads (LPTs) which makes it difficult to ensure performance isolation of the HPT.

For this experiment we use a worst-case scenario. We select 5 benchmarks as HPT (`ammp`, `art`, `mgrid`, `parser` and `vpr`) that require many ways to achieve their maximum IPC (large $w_{90\%}$ value). For benchmarks with low cache requirements, it is less challenging to attain the target IPC. We use the *4C-2* configuration, since as the number of LPTs is high it is more difficult to ensure the target IPC for the HPT. As LPTs we generate 3 groups of threads: we form the H group with cache hungry applications (`apsi+facerec+galgel`), the S group with applications with small working sets (`crafty+gzip+vortex`) and the L group with applications that do not benefit from more cache space (`equake+mesa+mcf`).

In Figure 6 we show the relative IPC of the HPT for the different workloads and different target percentages. On the x-axis, the target percentage of the full speed of the HPT is given, ranging from 20 to 90 percent. For each HPT and type of the LPTs, we give the achieved relative IPC for the HPT (measured on the y-axis). For example, the first bar corresponds to `ammp` when mixed with the H group `apsi+facerec+galgel`. The target relative IPC is 20% and we reach 32%.

Note that the number of assigned ways is discrete and, as a consequence, we cannot always force an exact target IPC. Instead, our mechanism assigns to the HPT the minimum number of ways needed to be above the target IPC, which still ensures the target IPC for the HPT. Some benchmarks already exceed the target IPC with just one reserved way in the L2 cache. This is the case of `mgrid` that runs at 69% of its full speed with 1 way. Note that, over the entire range of different workloads and target percentages, the achieved IPC follows the trend of the target IPC.

If we consider that we accomplish the target IPC for the HPT when we obtain an IPC with a margin of error of 5%, then we have a success rate of 89.3%, `parser` being the only benchmark that does not succeed in some cases (success rate of 47% due to its poor prediction accuracy). We also observe that when the LPTs have high cache requirements (type H), it is more difficult for the HPT to obtain its target IPC than when LPTs are type L or S. When the target performance is not achieved, the HPT is 6.38% under the objective IPC on average.

There are two main reasons that justify why some HPTs are below their target performance. First, OPACU methodology shows low accuracy in IPC predictions for some benchmarks like `parser`, that has an IPC prediction error of 9.8%. Thus, the predicted full speed IPC can differ from the real one. Our mechanism assigns to the HPT the minimum number of ways needed to reach the target IPC, which is computed as a given percentage of the predicted full speed IPC. If this prediction is inaccurate, the decision can be wrong. To solve this problem, the accuracy of OPACU methodology should be improved. The second reason why some HPTs are below target is the shared bandwidth to access the different L2 cache banks. In some situations the number of ways assigned to the HPT is enough to reach the target IPC (the decision is correct), but bus contention is too high and performance drops. We could use a *bandwidth arbiter* as in (24; 22) to overcome this situation and reserve a fraction of bandwidth to the HPT.

Figure 7 shows the results of our framework when the full speed of the HPT is known beforehand. This performance could be given by the user or obtained with an improved version of the OPACU runtime mechanism. Also, it has been shown (11) that multimedia applications have approximately the same performance for different instances of the same application. Thus, in multimedia applications the user knows beforehand the full speed of the application and can provide it to our framework.

We evaluate a four-core architecture with a 1MB 16-way L2 cache. As in the previous experiment, some benchmarks already exceed their target IPC with just one reserved way (`art` reaches 76% of its full speed with one way). Here, FlexDCP achieves the target IPC 94% of the time and the obtained IPC is much closer to the target IPC. For the remaining 6% of the time, our framework is 1.48% under the target IPC on average. We note that `vpr` and `ammp` cannot reach the 90% of their maximum performance because they need more than 13 ways and we are assuming that each application has at least one reserved way of the L2 cache (that is, there are not enough resources in the architecture).

4.2 Ensuring a Global Quality of Service

In this section we evaluate the performance of FlexDCP when optimizing an overall QoS metric. We compare our proposal with the best state-of-the-art dynamic cache partitioning mechanisms: *MinMisses* (26) that is the best policy in the literature improving throughput, and *Fair* (15) that is the best policy in the literature improving fairness. *MinMisses* estimates the number of misses of each running application for all cache configurations and selects the L2 cache partition that minimizes the total number of misses. Instead of minimizing the total number of misses, *Fair* forces all threads to have the same increase in percentage of L2 misses, trying to equalize the statistic $X_i = \frac{misses_{shared_i}}{misses_{alone_i}}$ of each thread i.

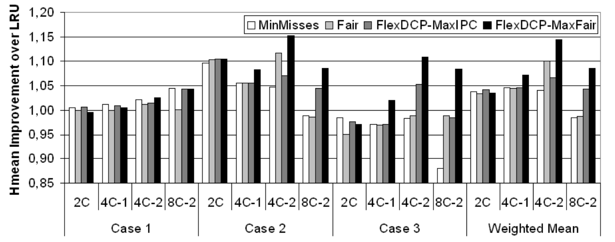


Figure 8. Hmean speed up over LRU when optimizing different QoS metrics.

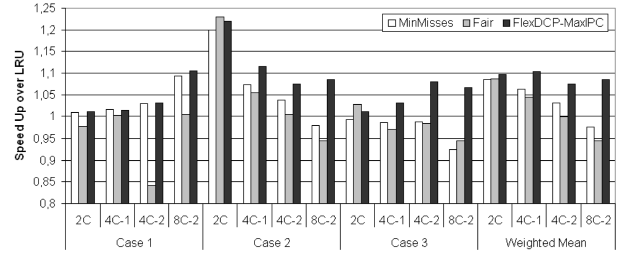


Figure 9. Throughput improvement of MinMisses, Fair and FlexDCP-MaxIPC over LRU.

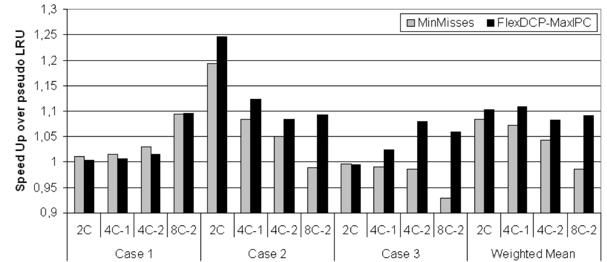


Figure 10. Average speed up over pseudo LRU when optimizing throughput.

Optimizing Fairness. By using predicted IPCs, we can decide to maximize any global QoS metric related to IPC. A relevant goal in some environments like high performance servers is to have fairness among threads. Several metrics have been used to measure fairness, like weighted speed up or the harmonic mean of relative IPCs. In this paper, we show results for the latter as it has been shown to provide better fairness-throughput balance than weighted speed up (18). In any case, our results for weighted speed up follow the same trends than for harmonic mean. We compute the relative IPC as $\frac{IPC_{CMP}}{IPC_{alone}}$. We denote our proposal *FlexDCP-MaxFair* as we maximize fairness. We compare our proposal with LRU, *MinMisses*, *Fair* and *FlexDCP-MaxIPC*.

To evaluate our proposals, we randomly generate 16 workloads belonging to each case for the four selected configurations[4] (48 workloads per configuration). Average improvements do consider the distribution of workloads among the three groups. We denote this mean *weighted mean*, as we assign a weight to the speed up of each case depending on the distribution of workloads from Table 3. For example, for the *2C* configuration, we compute the weighted mean improvement as $0.48 \cdot x_1 + 0.41 \cdot x_2 + 0.11 \cdot x_3$, where x_i is the average improvement in Case i.

Figure 8 shows the average Hmean improvement of all policies over LRU for the four configurations. We observe that for all processor/cache setups, *FlexDCP-MaxFair* outperforms the other proposals on average. For the 4-core configurations *FlexDCP-MaxFair* outperforms *Fair* by 3.5% and MinMisses by 6.5%. It is interesting to note that as the number of cores and cache size increase, the Hmean improvement of *FlexDCP-MaxFair* over previous proposals also increases, outperforming *Fair* by 10.1% and *MinMisses* by 10.3% on average in the largest configuration (*8C-2*).

All algorithms have similar results in Case 1. This is intuitive as in this situation there is little room for improvement as all threads fit in cache. In Case 2, *FlexDCP-MaxFair* improves previous approaches between 8.2% and 15.2%. As the number of cores increases, *MinMisses* and *Fair* have more difficulties in finding out the optimal partition for fairness. Instead, in configuration *8C-2*, *FlexDCP-MaxFair* achieves an improvement of 10.8% over *Fair* and 9.8% over MinMisses. In Case 3, *MinMisses* and *Fair* present performance degradations with respect to LRU because of the asymmetry between the cache requirements of applications. As a result, *MinMisses* has worse average fairness than LRU (4.6% on average). The same happens with *Fair*, which has 2.6% worse performance than LRU. By using IPC predictions *FlexDCP-MaxFair* instead obtains better results than LRU, 8.6% in the 8-core configuration and 10.9% in the 4-core configurations.

Optimizing Throughput. Next, we analyze the results of FlexDCP when it maximizes throughput. We denote this proposal *FlexDCP-MaxIPC*, as the metric to optimize is throughput. We simulate *MinMisses*, *Fair* and *FlexDCP-MaxIPC* with the same 48 workloads that we selected for the fairness results. Figure 9 shows the average speed up over LRU for these mechanisms. *FlexDCP-MaxIPC* provides the best performance for all cache configurations. In 83.2% of the workloads, *FlexDCP-MaxIPC* outperforms the throughput obtained by *MinMisses*, which means that performance improvements are consistent among workloads.

Figure 9 shows that the performance benefits of *MinMisses* decrease with the increase in the number of cores and associativity, obtaining 2.4% less throughput than LRU in configuration *8C-2*. In the *2C* configuration it improves 8.5% over LRU, while in *4C-1* and *4C-2* these benefits decrease to 6.2% and 3.1% respectively. *Fair* obtains even worse results than *MinMisses* in all configurations. Instead, *FlexDCP-MaxIPC* has a more consistent throughput improvement over LRU: 9.7% (*2C*), 10.3% (*4C-1*), 7.5% (*4C-2*), and 8.4% (*8C-2*). Figure 8 shows that performance improvements over LRU and *MinMisses* are not at the expense of fairness, as we improve both in fairness and throughput.

Interaction with real pseudo LRU implementations. One of the key aspects of any QoS framework to be considered by industry is that it has to work with replacement policies implemented in real processors. For highly associative caches, implementing true LRU replacement becomes complex and with a high hardware cost. As a consequence, current high performance processors implement other simpler replacement algorithms in the L2 cache with similar performance to LRU (9). For example, the Sun UltraSPARC T2 (2) has a shared 4Mbyte 16-way associativity L2 cache with pseudo LRU replacement, which has a used-bit scheme to implement a Not Recently Used (NRU) replacement. The used bit is set to one each time a cache line is accessed or when initially fetched from memory. For a given cache set, if setting the used-bit causes all used bits to be set to one, the remaining used bits are cleared instead. On a miss, the L2 looks for the first line in that set with a used-bit set to zero, which is chosen as the evicted line.

We propose to partition a shared L2 cache with NRU replacement algorithm extending columnization, which was proposed to partitioning caches with LRU replacement algorithm (5). We assign an initial used-bit mask (UBM) for each thread that sets to one the ways owned by other threads and sets to zero its owned ways. Thus, threads can evict only lines from their owned ways. Extending columnization for NRU replacement algorithm is mechanical and we do not give all implementation details for the sake of simplicity. Exploring the interaction with other replacement algorithms is part of our future work.

[4] Except Case 1 in configuration *8C-2*, as only one workload belongs to this group.

Table 4. Functionalities offered by the different frameworks.

Framework	Performance/resource translation	Provides individual QoS	Provides hybrid QoS	Provides global QoS	Fairness	Throughput
MinMisses (26)	×	×	×	✔	+	++
Fair (15)	×	×	×	✔	++	+
VPC (24)	×	✔	×	×	−	−
Guo et al. (12; 6)	×	✔	✔	×	−	−
FlexDCP	✔	✔	✔	✔	+++	+++

Next, we show that our QoS framework is compatible with NRU replacement algorithm. Figure 10 shows the average speed up over NRU when optimizing a global QoS metric like throughput. We evaluate *MinMisses* and *FlexDCP-MaxIPC* with the workloads that were selected in Section 3.2. Neither the SDHs nor the PPRs will provide results as accurate as with LRU. Even though, speed ups are nearly the same than the ones obtained with LRU. *MinMisses* presents diminishing returns as the number of core increases, with an average improvement of 4.6% over NRU. *FlexDCP-MaxIPC* has more consistent results with an average improvment of 9.6% over NRU. The results when optimizing Hmean and individual QoS present the same trend with Figures 6, 7 and 8.

4.3 Putting it all together

FlexDCP is the only framework providing enough flexibility to provide individual or global QoS to applications. FlexDCP can maximize overall QoS metrics like harmonic mean of relative IPCs, weighted speed up or throughput, outperforming LRU and previous throughput- or fairness-oriented DCP algorithms. The performance results presented in this section prove that using performance projections to decide cache partitions is more adequate and leads to better performance than previous proposals guided by miss rates.

5. Related Work

QoS-aware architectures. Some previous efforts focus on ensuring QoS in multithreaded architectures. Cazorla et al. introduce a mechanism to force predictable performance in SMT architectures (4). They manage to run time-critical jobs at a given percentage of their maximum IPC. To attain this goal, they need to control all shared resources of the SMT architecture, while we work with a CMP architecture.

Concerning CMP architectures, Rafique et al. propose to manage shared caches with a hardware cache quota enforcement mechanism and an interface between the architecture and the OS to let the latter decide quotas (27). However, this proposal cannot guarantee individual QoS. Nesbit et al. introduce Virtual Private Caches (VPC) (24), which consist of an *arbiter* to control cache bandwidth and a *capacity manager* to control cache storage. They show how the arbiter allows meeting QoS performance objectives or fairness. However, the authors do not discuss how to decide resource assignments. A similar framework is presented by Iyer et al. (12), where resource management policies are guided by thread priorities. Individual applications can specify their own QoS target (e.g. IPC, miss rate, cache space) and the hardware dynamically adjusts cache partition sizes to meet their QoS targets. An extension of this work with an admission mechanism to accept jobs is presented in (6). However, the authors claim that IPC is not suitable to specify a QoS target because IPC is not easily convertible into resource allocation. In this paper, FlexDCP attains this objective of converting IPC into resource assignment.

Lee et al. present METERG QoS system, which provides QoS in a soft real time scenario (16). However, in this framework, the developer needs to run the application in the system before to guarantee a QoS in future executions. With our framework, no profiling information is needed to guarantee a QoS.

Table 4 compares the functionalities that previous proposals offer with FlexDCP. In this table, we use the following symbols: +++ (very high), ++ (high), + (medium), − (low, equivalent to LRU), ✔ (feature supported), × (feature not supported). FlexDCP is the first framework to cover all the necessary steps to convert performance and QoS requirements into resource assignments. Furthermore, the flexibility of the framework allows ensuring all concepts of QoS, giving the best performance when optimizing global QoS metrics like fairness or throughput.

Dynamic cache partitioning. Traditional eviction policies are demand-driven and tend to give more space to the application that has more accesses to the cache hierarchy. Previous work has proposed to partition shared caches, assigning more cache space to the applications that do a better use of this space. Static and dynamic cache partitioning algorithms monitor the L2 cache accesses and decide a partition in order to maximize throughput (5; 26; 32; 29) or fairness (15). These proposals decide cache partitions at a *way granularity* based on collected data. However, these proposals use indirect metrics of performance like the number of misses to guide decisions, and they cannot guarantee individual QoS.

Stack Distance Histogram. Common eviction policies such as LRU have the *stack property* (19). This property allows building Stack Distance Histograms (SDH) which can be obtained during execution by running the thread alone in the system (5) or by adding some hardware counters that profile this information (26; 32). Qureshi et al. presented a low-overhead circuit to measure SDHs using an auxiliary tag directory, which is a separate copy of the L2 tags (26). Nearly all cache partition schemes at way granularity are based on SDHs. Some authors have used SDHs to improve the management of main memory and reduce the number of page faults (36).

Other Related work. Hsu et al. evaluate different cache policies in a CMP scenario (10). They show that none of them is optimal among all benchmarks and that the best cache policy varies depending on the performance metric being used. Thus, they propose to use a thread-aware cache resource allocation. In fact, their results reinforce the motivation of our paper: we need to have a flexible framework that allows optimizing different performance metrics and being able to ensure a QoS.

6. Conclusions

In this work, we propose FlexDCP, a new framework that allows the OS to guarantee a QoS for each application running in a CMP architecture. Instead of using indirect measures of the performance, FlexDCP uses direct estimations of the performance of each thread for different cache configurations to decide cache quota assignments, with less than 1KB of storage cost per core. These estimations enable our framework to control the performance of individual applications when executed in a workload, ensuring an individual QoS. In addition, this framework provides higher flexibility than previous proposals as it allows the OS to optimize either fairness, total throughput, or any other metric.

Simulation results show that FlexDCP is able to force applications to run at a certain percentage of their maximum performance, which is required in real-time environments. We manage to reach the objective performance in 94% of the cases considered, being

1.48% under the objective for remaining cases. When optimizing for a global QoS metric like fairness or throughput, FlexDCP obtains the best performance in all metrics. In an eight-core architecture, *FlexDCP-MaxFair* obtains an average 10.1% improvement over *Fair* in fairness, while *FlexDCP-MaxIPC* obtains an average 11.2% improvement over *MinMisses* in throughput. Finally, we showed that FlexDCP also works with pseudo LRU replacement algorithms currently implemented in processors like the Sun UltraSPARC T2.

FlexDCP provides a platform that can also be used with parallel applications. In single process-multiple data applications, all the processes execute the same code on different data sets and use synchronization primitives to coordinate their work. Thus, FlexDCP framework can be used to estimate the performance of each process between communication primitives. In the case of parallel applications in which threads concurrently work on the same data, the parallel application can be seen as a whole accessing the shared cache. With that goal, bit masks should be assigned to processes instead of cores. Future work includes evaluating the impact of these extensions on the FlexDCP framework.

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN-2004-07739, TIN2007-60625 and grant AP-2005-3318, by the HiPEAC European Network of Excellence and by the SARC European Project. Authors would also like to thank C. Acosta, A. Falcon, D. Ortega, O. J. Santana and J. Vermoulen for their work in the simulation tool. We also thank F. Cabarcas, I. Gelado, D. A. Jiménez, A. Rico, J. E. Smith and C. Villavieja for their technical comments on earlier drafts of this paper and the reviewers for their helpful comments.

References

[1] ARM920T. Technical Reference Manual. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0151c/ARM920T_TRM1_S.pdf`.

[2] UltraSPARC T2. Supplement to the UltraSPARC Architecture 2007. `http://opensparc-t2.sunsource.net/specs/UST2-UASuppl-current-draft-HP-EXT.pdf`.

[3] D. P. Bovet and M. Cesati. *Understanding Linux kernel*. O'Reilly, 3rd edition, 2005.

[4] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE ToC*, 55(7):785–799, 2006.

[5] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *Design Automation Conference*, 2000.

[6] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *MICRO*, 2007.

[7] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.

[8] L. C. Heller and M. S. Farrell. Millicode in an IBM zSeries processor. *IBM J. Res. Dev.*, 48(3-4):425–434, 2004.

[9] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., 3rd edition, 2002.

[10] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *PACT*, 2006.

[11] C. J. Hughes, P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan. Variability in the execution of multimedia applications and implications for architecture. In *ISCA*, 2001.

[12] R. R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. R. Hsu, and S. K. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, 2007.

[13] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. C. S. Jr, and J. Emer. Adaptive insertion policies for managing shared caches on cmps. In *PACT*, 2008.

[14] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA*, 2004.

[15] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.

[16] J. W. Lee and K. Asanovic. METERG: Measurement-based end-to-end performance estimation technique in QoS-capable multiprocessors. In *RTAS*, 2006.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[18] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*, 2001.

[19] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[20] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero. Explaining dynamic cache partitioning speed ups. *IEEE CAL*, 2007.

[21] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero. Online prediction of applications cache utility. In *IC-SAMOS*, 2007.

[22] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO*, 2007.

[23] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO*, 2006.

[24] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA*, 2007.

[25] K. J. Nesbit, M. Moreto, F. J. Cazorla, A. Ramirez, M. Valero, and J. E. Smith. A framework for managing multicore resources. *IEEE Micro, special issue on Interaction of Computer Architecture and Operating System in the Many-core Era*, 38(3), 2008.

[26] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.

[27] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *PACT*, 2006.

[28] M. J. Serrano, R. Wood, and M. Nemirovsky. A study on multistreamed superscalar processors. Technical Report 93-05, UCSB, 1993.

[29] A. Settle, D. Connors, E. Gibert, and A. Gonzalez. A dynamically reconfigurable cache for multithreaded processors. *Journal of Embedded Computing*, 1(3-4), 2005.

[30] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 2003.

[31] J. E. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann Publishers Inc., 2005.

[32] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA*, 2002.

[33] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA*, 1995.

[34] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. FAME: Fairly measuring multithreaded architectures. In *PACT*, 2007.

[35] T. Y. Yeh and G. Reinman. Fast and fair: data-stream quality of service. In *CASES*, 2005.

[36] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS*, 2004.