# A Hardware Approach to *Fairly* Balance the Inter-thread Interference in Shared Caches

Vicent Selfa, Julio Sahuquillo, *Member,IEEE,* Salvador Petit, *Member,IEEE,* and María E. Gómez, *Member,IEEE,*

**Abstract**—Shared caches have become the common design choice in the vast majority of modern multi-core and many-core processors, since cache sharing improves throughput for a given silicon area. Sharing the cache, however, has a downside: the requests from multiple applications compete among them for cache resources, so the execution time of each application increases over isolated execution. The degree in which the performance of each application is affected by the interference becomes unpredictable yielding the system to *unfairness* situations.
This paper proposes Fair-Progress Cache Partitioning (FPCP), a low-overhead hardware-based cache partitioning approach that addresses system fairness. FPCP reduces the interference by allocating to each application a cache partition and adjusting the partition sizes at runtime. To adjust partitions, our approach estimates during multicore execution the time each application would have taken in isolation, which is challenging. The proposed approach has two main differences over existing approaches. First, FPCP distributes cache ways incrementally, which makes the proposal less prone to estimation errors. Second, the proposed algorithm is much less costly than the state-of-the-art ASM-Cache approach. Experimental results show that, compared to ASM-Cache, FPCP reduces unfairness by 48% in four-application workloads and by 28% in eight-application workloads, without harming the performance.

**Index Terms**—Cache partitioning; multi-cores; fairness; progress; slowdown; execution time in isolation

## 1 Introduction

SHARED caches can be found in the vast majority of modern multi-core and many-core processors. The main reason is that *cache sharing* improves throughput for a given silicon area. As a consequence, recent microprocessors incorporate shared caches in almost all, if not all, levels of the cache hierarchy. In this regard, all cache levels (e.g. L1, L2 and L3) are shared in simultaneous multithreading (SMT) processors, e.g. the multi-core IBM Power8 processor [1] and the many-core Knights Landing Intel Xeon Phi [2]. The benefits of cache sharing are also exploited in the embedded market, e.g., the L2 cache in the ARM Cortex-A53 processor [3]. Cache sharing, however, introduces interference when the co-running threads dynamically contend for cache resources. Consequently, the performance of individual applications can be worse than when executed alone, depending on how severe is the contention.

This causes an *unfairness* problem, which is a major concern in current CMP design, since *unfairness* causes critical undesirable behaviors: i) it makes execution time unpredictable, which complicates the analysis of both hardware and software implementations, ii) it complicates priority-based Operating System (OS) scheduling, and iii) it enables denial of service attacks. Despite this, unfairness is still an unsolved problem in current microprocessors.

The unfairness of a system depends on the *progress* of the running applications. This *progress* metric is defined, per application, as the quotient between its execution time when executed with co-runners and its execution time when executed in isolation. Slowdown is the inverse of progress, and both metrics can be used to measure how interference degrades performance. Using the progress metric, unfairness is formally defined as the quotient between the progress of the most progressing application and the progress of the least progressing application [4]. Consequently, a system is considered to be completely fair when all the tasks in the system experience the same *progress*, so unfairness is equal to one [5], [6], [7], [8].

To illustrate the unfairness problem in a real system we measured the *progress* of four applications running concurrently on a recent Intel multicore processor, the Xeon E5 2658 v3. The Intel Xeon E5 2658 v3 supports the execution of 24 applications (i.e. 12 SMT-2 cores) at the same time and features a 30MB L3 shared cache. The L2 caches, on the other hand, are shared among the 2 hardware threads that run in each core when hyperthreading (i.e. SMT) is enabled. In order to avoid intra-core interference, each application was allocated to a different core, disabling the SMT mode. Figure 1 shows the progress that each application (`libquantum`, `lu`, `omnetpp` and `sphinx3`) experienced with respect to individual execution. It can be appreciated that `sphinx3` is the most progressing application, with a progress rate of 82% (i.e. 22% slowdown) and `libquantum` is the least progressing application, with a progress rate of 57% (75% slowdown). The unfairness is, therefore, 44% (82% / 57% $-1=0.44$). Notice that the unfairness problem would exacerbate if the L1 and L2 had been shared, which could happen if two applications were allocated to the same core.

To address unfairness in modern multicore processors, this paper proposes *Fair-Progress Cache Partitioning* (FPCP), a low-overhead hardware-based cache partitioning approach

• *V. Selfa, J. Sahuquillo, Salvador Petit and M. E. Gómez are with the Department of Computing Engineering, Universitat Politècnica de València, Spain.*
*E-mail: {viselol,jsahuqui,spetit,megomez}@disca.upv.es*

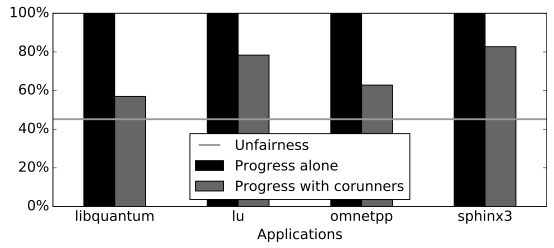


Fig. 1. Progress and unfairness for 4 applications running concurrently in an Intel Xeon E5 2658A during 120 seconds.

that reduces cache interference by allocating a cache partition to each application and adjusting its size dynamically at runtime, since the cache requirements of each application vary during its execution. Thus, FPCP acts periodically, modifying the number of ways allocated to each partition, giving more cache space to applications suffering more slowdown. Section 5 explains the proposal in detail.

A key characteristic of FPCP is that the number of cache ways provided to a given application can only vary in one-unit between two consecutive intervals. This particularity makes the hardware simpler, the mechanism more resilient to deviations in the estimations and, as experimental results will show, allows the system to achieve the best cache distribution. However, the slowdown an application suffers is unknown at runtime, so a key challenge to deal with fairness is the estimation of the execution time each application would experience in isolation. To deal with this issue we need a performance model that, taking multi-core performance and inter-thread interference as inputs, provides us with accurate performance estimations of isolated execution as output.

Two approaches have been recently proposed to estimate isolated execution performance. The first one, *Per-Thread Cycle Accounting* (PTCA) [9], identifies at run-time the cache misses that would have not occurred in isolation (i.e. inter-thread misses). Then, the amount of cycles the reorder buffer (ROB) is blocked due to these misses is subtracted from the total execution time to obtain an estimation of the execution time the application would experience if executed without co-runners. The other, *Application Slowdown Model* (ASM) [10], is conceptually similar. However, ASM uses the Cache Access Ratio ($CAR$), cache accesses per time unit, as a proxy for performance. In this work, both approaches have been implemented in order to check and evaluate our proposal. Experimental results show that, in our implementation, the ASM approach is a bit more accurate than PTCA. Section 4 explains the differences between this two approaches and compares them.

Finally, Section 6 compares FPCP with two state-of-the-art cache partitioning mechanisms, Utility Cache Partitioning (UCP) [11], by Qureshi and Patt, and ASM-Cache, a recent proposal by Subramanian *et al.* [10] that works on top of the ASM performance model. While both help to reduce unfairness, they present several drawbacks: i) since finding the optimal partitioning is a NP-hard problem [12], both UCP and ASM-Cache employ a $\mathcal{O}(n^2)$ greedy algorithm, compromising scalability; ii) they strongly depend on the accuracy of the estimation of the execution time in isolation, so errors in this estimation can have a big impact on application progress and system unfairness, and iii) both rely on the cache replacement policy obeying the stack property [13] (as with LRU) and in being able to track the number of hits in each frame of the replacement policy stack. In contrast, FPCP avoids these limitations by featuring an incremental cache partitioning algorithm, which is both less complex and more resilient to estimation errors.

This paper makes the following contributions.

- We characterize the applications from SPEC CPU2006 and NAS benchmark suites according to their relationship between progress and shared cache interference, analyzing how unfairness may be affected depending on both the co-running applications and the available cache resources.
- We implement and compare two different models to estimate isolated execution performance, PTCA and ASM, concluding that ASM is slightly more accurate.
- We propose FPCP, a simple, cost-effective and scalable cache partitioning mechanism that improves system fairness regardless of the number of contending applications.
- We show that FPCP achieves better fairness than two state-of-the-art cache partitioning mechanisms, UCP and ASM-Cache, across a wide range of workloads and system configurations.

The remainder of this work is organized as follows. Section 2 discusses the related work. Section 3 analyzes the relationship between interference, progress and fairness. Section 4 discusses the way progress is estimated at runtime. Section 5 presents the repartitioning approach, and Section 6 evaluates the proposal. Finally, in Section 7 some concluding remarks are drawn.

## 2 Related Work

Some research works partition shared last level caches (e.g. L2 or L3) focusing on performance, QoS or fairness. The specific implementation of each proposal can be hardware-based or software-based, and those that are hardware-based can be evaluated using real hardware or in a simulation environment.

Regarding approaches addressing QoS in CMPs with shared caches, in [14] Iyer presents the CQoS cache management framework, which provides prioritized service to multiple heterogeneous threads sharing a cache structure. CQoS relies on hardware support to enforce priorities among the memory access streams issued by the different threads. In [15], Chang and Sohi use multiple time-sharing cache partitions to guarantee QoS among threads. They propose a QoS metric that modulates the allocated cache space for a given thread. Ubik [16] employs ZCaches [17], a radical change from traditional caches, to partition the cache. Their goal is to provide QoS while at the same time improving the performance of batch applications.

Other recent approaches also targeting QoS employ Intel Cache Allocation Technologies (CAT) [18]. Both Heracles [19] and Dirigent [20] focus on maximizing utilization in large-scale datacenters without affecting the user-perceived latency in latency critical applications. To do that, and among other approaches, they classify applications *a priori* as *batch* or *latency critical*, and use CAT to limit the amount of cache resources that batch applications can consume. Ginseng [21] is also based on CAT, but focuses on cloud computing providers that rent virtual machines. It uses a market-driven auction system to partition the LLC into non-overlapping partitions depending upon how much each guest is willing to pay and how that affects the rest. Note that approaches using CAT are controlled from software, while our approach works at hardware level, with a granularity orders of magnitude smaller. Additionally, the amount of information available to the operating system and userland software is limited to public-exposed performance counters, while a hardware-level approach has access to much more information.

In [22], Hsu *et al.* demonstrate that modern CMPs require policies to adequately distribute the cache space, since LRU is not sufficient to meet neither performance nor fairness goals. They also formalize the different aspects that characterize a partitioning policy and conclude that policies targeting fairness are often near optimal, performance-wise. Most techniques aimed at achieving fairness, which is the goal of this work, are software-based and rely on OS scheduling [23], [24], [25]. Others, like [5], try to achieve system fairness by dynamically adapting the rate at which different cores inject requests to the memory subsystem. Fedorova *et al.* [26] propose a software-based technique that regulates OS time slices to prevent the IPC of a given thread from being lower than estimated using the thread *fair miss rate*. The fair miss rate of a thread is defined as the miss rate that the thread experiences when the shared cache is equally distributed among concurrent threads. The approach followed in [27] is based on changing the LRU policy to focus on fairness among cores by penalizing the core with highest IPC in favor of the others. The work in [28] evaluates several static partitioning approaches to achieve fairness on an Ivy Bridge architecture. It employs a theoretical approach to model application LLC features. In [29], Kim *et al.* try to improve system fairness by partitioning the shared L2 space without requiring any OS modification. In this work, they analyze four metrics and their correlation with system fairness. However, three of them require offline profiling, making them impractical. We considered the last metric as a base for our proposal, but we found that it strongly depends on the characteristics of the memory hierarchy of the CMP, which makes it unsuitable.

Some approaches such as [30], [31], [32], [33], [34], pursue to improve the raw cache performance. An interesting idea is the work by Qureshi and Patt [11], Utility Cache Partitioning (UCP), which partitions the cache to minimize cache misses and maximize cache throughput. Other works focusing on throughput, like [35] and [36], require OS intervention. Vantage also targets performance but does not use traditional caches but ZCaches, which provide Vantage with more flexibility for partitioning the cache. While Vantage uses an adaptation of the Utility Cache Partitioning algorithm to distribute cache space, our proposal could also be adapted to work on top of it. Notice that while the cache partition policy proposed in this paper assumes that partitions are enforced by augmenting LRU cache replacement policy to allow way partitioning [14], other schemes such as those presented in [37] can be used.

A significant amount of work has been devoted to software-based cache partitioning approaches [38], [39], [40]. Most of them are based on page-coloring techniques that manipulate the address bits used to determine the cache set the data blocks are placed into. Coloris [41] targets QoS, and monitors the cache miss rates of running applications, re-partitioning the cache when miss rates exceed applications-specific thresholds. In [42], the authors propose a lightweight dynamic partitioning approach for the LLC that employs page coloring to improve performance. However, the main disadvantage of page coloring comes when there is frequent repartitioning, since memory pages must be copied to new locations to change the cache allocation, so is less versatile than a hardware mechanism, as the one presented in this paper.

## 3 Analysis of the inter-application cache interference

The interference an application suffers when contending for the shared resources with other applications has an unpredictable impact on its progress (Equation 1), and thus in the unfairness of the system, measured as in Equation 2. In Equation 1, $C_{t,multicore}$ is the measured execution time of task $t$ with less effective cache resources due to contention, and $C_{t,alone}$ is the estimated execution time task $t$ would experience were it executed without interference.

$$Progress_t = \frac{C_{t,alone}}{C_{t,multicore}} \tag{1}$$

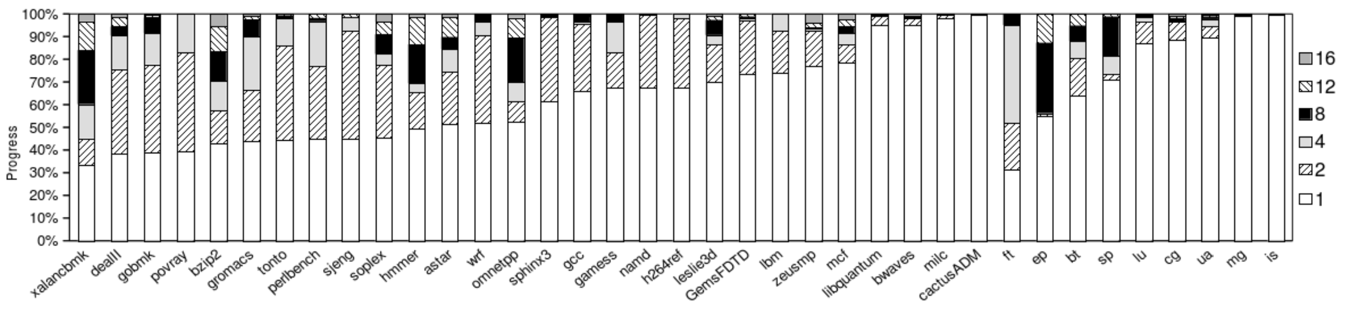
$$Unfairness = \frac{\max Progress_i}{\min Progress_j} \mid i,j \in Tasks \tag{2}$$

To help to understand the causes of unfairness, this section analyzes how the progress of each individual application is affected when the amount of assigned cache ways varies from 1 way to the total available, simulating other applications competing for the same cache resources[1].

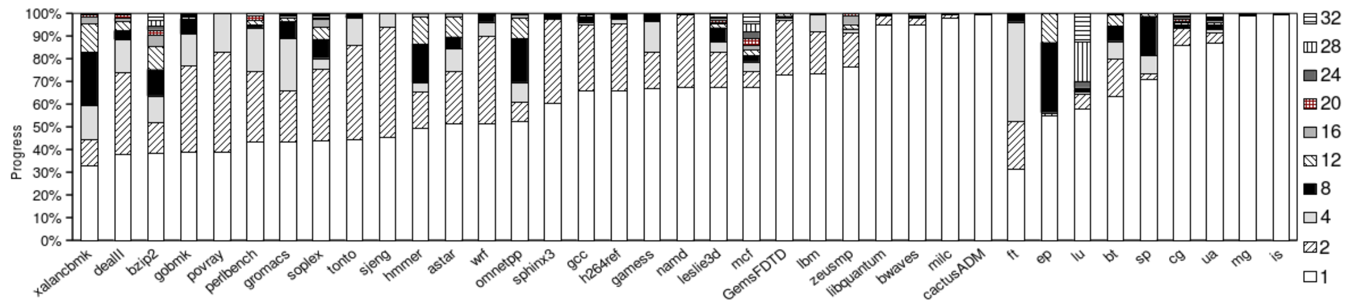
Since results depend on the cache geometry, to focus the analysis we first consider a 2MB-16w LLC cache. Figure 2a shows the progress rate (from 0% to 100%) achieved by the different applications (sorted by ascending progress with 1 cache way) when varying the number of assigned cache ways. For instance, `povray` achieves a progress rate of around 39% with just 1 cache way and requires 2 and 3 ways to increase its progress up to 82% and 100%, respectively. These results have been obtained using the simulation environment described in Section 6.

According to the number of ways required to achieve a significant (e.g. 80%) progress, three main categories of applications can be distinguished: *cache-insensitive progress*, *highly cache-sensitive progress* and *moderately cache-sensitive progress*. The former group contains those applications whose progress is barely affected by the number of cache ways assigned to the application, that is, a single cache way is enough to achieve significant progress. The second and

1. The progress of a task when running in the cache with a reduced number of ways, say $w$, is computed as the execution time of the task when it has the entire cache for itself (i.e. 16 ways and no interference) divided by the execution time taken in the constrained cache.



(a) 2MB-16w LLC



(b) 4MB-32w LLC

Fig. 2. Effect of the available number of ways on progress, sorted by 1-way progress, for applications of the SPEC 2006 and NAS benchmark suites.

third categories group those applications whose progress is sensitive to the number of ways. However, while applications in the second group require a high number of cache ways (e.g. half the number of the total cache ways), applications in the last group require a relatively low (e.g. from 2 to 4) number of ways to achieve significant progress.

Next, we illustrate how this information can be used to provide an overview analysis about system unfairness through two examples: one with low unfairness and another one with a high level of unfairness.

**Low-unfairness example.** Assume that `cactusADM` and `milc` run together. In this case, unfairness would not be a concern since both applications are classified as *cache-insensitive progress*, and the progresses of both applications would be higher than 97% regardless of the cache way distribution.

**High-unfairness example.** Assume now that the applications running together are `cactusADM` and `xalancbmk`. The former is an application with *cache-insensitive progress* while the latter is classified as *highly cache-sensitive progress*. Thus, while `cactusADM` only requires one way to achieve a notable progress, `xalancbmk` requires significantly more cache resources to have an adequate progress rate. However, due to its poor cache locality under the typical LRU replacement algorithm, which is the reason its progress is so cache insensitive, `cactusADM` uses around 10 out of 16 cache ways, leaving to `xalancbmk` only the remaining 6. Consequently, the progress of `xalancbmk` drops below 80%, which yields the system to an unfairness level of around 25%.

An analogous study has been performed for a 4MB-32w cache (see Figure 2b). It can be noticed that, although using the same sorting approach, the relative order of the workloads in Figure 2a and in Figure 2b is slightly different. This can occur when the working set fits in the larger cache or when the larger cache improves the hit ratio enough to significantly reduce cache trashing.

## 4 Analysis of Progress Estimation Approaches

FPCP employs auxiliary circuitry to estimate the execution time each application would have experienced if executed without co-runners, since this information is required to estimate the progress of the application, see Equation 1 in Section 3. Then, the progress results are used to select cache partition sizes. Therefore, if estimations are not accurate enough, it is likely that the cache partitioning will perform poorly.

There are are two recent approaches to estimate performance without co-runners: Per-Thread Cycle Accounting (PTCA) by Du Bois *et al*. [9] and Application Slowdown Model (ASM) by Subramanian *et al*. [10].

To implement and evaluate FPCP, we first implemented and compared the PTCA and ASM models to obtain progress estimations, based on the guidelines discussed in the original works. To make this paper self contained, some key guidelines are discussed below. Please, refer to the original work for further details.

Both approaches make use of an Auxiliary Tag Directory (ATD) per core, which is a structure that keeps track of what the status of the shared cache would have been if it were private to the core [11]. Note that if SMT is used and one wants to distribute cache resources per-application instead of per-core, then an ATD per thread is required and the mentioned models need to take into account the interference between threads at the shared cache levels above the LLC.

The key challenge lies on obtaining accurate estimates of performance in isolation by using information gathered during execution with co-runners. This can be done by subtracting the cycles an application makes no progress due to interference caused by co-runners from the concurrent execution time (see Equation 3, where $I_{t,multicore}$ represents the stall cycles due to interference).

$$C_{t,alone} = C_{t,multicore} - I_{t,multicore} \tag{3}$$

Using the ATD, PTCA identifies at run-time the LLC cache misses that would have not occurred in isolation (i.e. inter-thread misses). Then, the amount of cycles the reorder buffer (ROB) is blocked due to these misses is accounted as interference cycles.

The approach followed by ASM is conceptually similar. However, ASM uses the Cache Access Ratio to the LLC ($CAR$) as a proxy for performance. $CAR_{t,multicore}$ is obtained during execution with co-runners and it is defined as in Equation 4. On the other hand, $CAR_{t,alone}$ is estimated dividing the number of cache accesses to the LLC by the cycles elapsed minus the cycles lost due interference (see Equation 5). Notice that the fraction's denominator of the latter equation matches $C_{t,alone}$ when applying Equation 3.

$$CAR_{t,multicore} = \frac{\#LLC\_Accesses}{C_{t,multicore}} \tag{4}$$

$$CAR_{t,alone} = \frac{\#LLC\_Accesses}{C_{t,multicore} - I_{t,multicore}} = \frac{\#LLC\_Accesses}{C_{t,alone}} \tag{5}$$

Therefore, progress is defined in ASM as shown in Equation 6. Note that the original ASM paper estimates slowdown instead of progress but, since one is the inverse of the other, both can be used with the same aim.

$$Progress_t = \frac{CAR_{t,multicore}}{CAR_{t,alone}} = \frac{\frac{\#LLC\_Accesses}{C_{t,multicore}}}{\frac{\#LLC\_Accesses}{C_{t,alone}}} = \frac{C_{t,alone}}{C_{t,multicore}} \tag{6}$$

ASM uses a different method than PTCA to compute the interference cycles. Instead of tracking the time the ROB is stalled due to interference, they multiply the number of inter-thread misses (obtained with the ATD) by the average cache miss service time. Note that both ASM and PTCA provide mechanisms to separate and identify interference coming from different parts of the system so, while in this work we are only targeting LLC-originated unfairness, additional sharing policies, orthogonal to our proposal, could be implemented in other shared parts of the system, like the memory controller or the NoC [43], [44], [45] to further reduce unfairness.

We compared the accuracy of both ASM and PTCA and we found that, in our experimental setup, ASM was slightly more accurate than PTCA. Thus, results will be presented only with the ASM model. Figure 3 shows the average and the standard deviation of the estimation error across the studied workloads varying the number of applications running concurrently.

## 5 FPCP Partitioning Approach

As mentioned above, a system is totally fair if all the co-running tasks progress at the same pace with respect to isolated execution. The proposed partitioning approach pursues to minimize system unfairness by narrowing progress differences among co-executing tasks. FPCP gathers interference data during multicore execution at regular intervals,

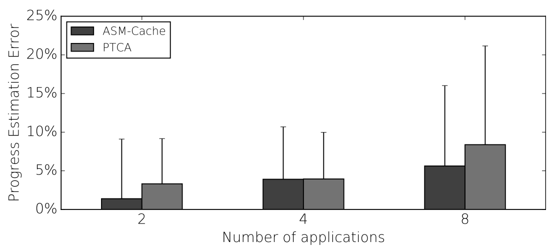


Fig. 3. Progress estimation error with ASM and PTCA.

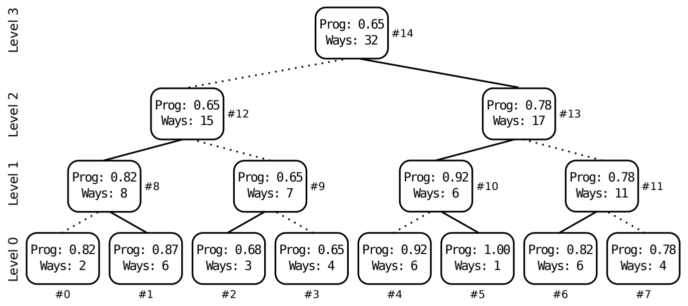


Fig. 4. Logical tree structure used by FPCP.

and then estimates progress at the end of each interval according to Equation 6 and distributes cache ways.

We evaluated different interval lengths and found that the best results for FPCP were obtained with intervals of 5K misses. The reason we use misses to define the interval length instead of cycles is that this way responsiveness is increased during execution phases with lots of cache accesses.

Next, we analyze the scalability of FPCP with the number of cores, we discuss the reasons behind using binary trees as underlaying structure, and finally, we estimate the proposal overhead.

### 5.1 Algorithm and hardware implementation

FPCP implements a hardware tree-based algorithm, which requires little extra logic [46]. The set of *n* applications running in the processor is subdivided recursively until the subsets have only two applications, and a *binary tree* with the *n* applications at the leafs is constructed. Figure 4 shows the resulting tree for 8 applications sharing a cache in an 8-core CMP.

Each leaf node contains the number of ways assigned to the corresponding application as well as the Modified Moving Average (MMA) [47] of its progress (see Equation 7).

$$MMA_i = MMA_{i-1} + f(Progress_{i-1} - MMA_{i-1})|_{f=0.1} \tag{7}$$

The reason why we use the MMA is that it approximates a conventional moving average, but only requires the previous average and the progress from the current interval to compute the average for the next. In addition, it retains enough previous information to make solid partitioning decisions while allowing a fast reaction time to changes in the workload behavior.

Each non-leaf node, on the other hand, stores i) the total number of ways assigned to its children and ii) the minimum progress (MMA) between its children. Note that, to ease the understanding of the example, in Figure 4 we use

```
foreach $n \in N_{level}$ do
    if $n.left.progress < n.right.progress$ then
        $least \leftarrow n.left$
        $most \leftarrow n.right$
        $n.prog \leftarrow n.left.progress$
    else
        $least \leftarrow n.right$
        $most \leftarrow n.left$
        $n.prog \leftarrow n.right.progress$
    if $n.ways == most.ways + least.ways$ then
        if $most.ways > 1$ then
            $most$ gives 1 way to $least$
    else if $n.ways > most.ways + least.ways$ then
        $least$ gains 1 way
    else
        if $most.ways > 1$ then
            $most$ loses 1 way
        else
            $least$ loses 1 way
```

Fig. 5. FPCP algorithm.

dashed lines to indicate which child node has the minimum progress.

Each Level $i$ in the tree except Level 0 has an associated number of intervals $I_i$, where $i$ indicates the level depth. Every $I_i$ intervals the algorithm listed in Figure 5 is applied to the nodes in Level $i$ ($i \geq 1$). For each node in that level, the minimum progress between its children is determined and stored in the node by checking *only* the lower level. Then, there are three possible cases: ① The node has the same number of ways as its children combined. When this occurs, the most progressing child relinquishes a way (provided it has more than one) in favor of the least progressing child. ② The node has more ways assigned than its children. In this case no child has its ways reduced, but the least progressing node gains a way. ③ The node has less ways assigned than its children. If this happens, then no child receives a way, and a way is subtracted from the most progressing child, if possible, or its brother, if not.

In Figure 4, node `#8` is an example of case ①. When the algorithm is applied to this node, a way is transferred from node `#1` to `#0`. On the other hand, node `#11` is an example of case ②. It has 11 assigned ways, but one of these ways is not assigned either to node `#6` or node `#7`. This can happen if node `#10` gave a way to node `#11` in a previous application of the algorithm in the upper level (node `#13`). So, when the algorithm is applied to `#11`, `#7` gains a way, and `#6` is not affected. Finally, node `#10` is an example of case ③, since it has less ways assigned than its combined children. Again, this situation can occur due to a previous application of FPCP in an upper level.

From now on, we assume $I_1 = 1$, $I_2 = 4$, and $I_3 = 8$, which are the values we used to obtain the experimental results in Section 6. Therefore, way transfer at Level 1 will occur every 1×5K misses; at Level 2 every 4×5K misses, and at Level 3 every 8×5K misses. Additionally, progress is computed every 1×5K misses. While we transfer ways one at a time, this value can be increased, but there is a trade-off between responsiveness and a higher penalty due estimation errors. In addition, a threshold could be used to only transfer ways if the progress difference is considered significant.

## 5.2 Rationale for using binary trees as the underlying structure

FPCP 's goal is to ensure that applications progressing the most relinquish cache resources, and that the freed cache resources are assigned to those applications progressing the least. A simple approach could be to only transfer resources from the most progressing application to the least progressing application, since finding the minimum and maximum elements in a set has a $\mathcal{O}(n)$ cost, being $n$ the size of the set. However, the benefits of this approach do not scale as the number of applications increases, since we are considering only two applications and ignoring the rest. The distribution of cache resources would be, therefore, not responsive enough. Other approach could be to sort the applications by progress and adjust all the partitions according to this information. Although the idea seems appealing, the sorting cost is $\mathcal{O}(nlog(n))$, which could make prohibitive the hardware implementation cost. Using a binary tree and making updates by levels maintains the benefits of potentially exchanging ways between all the applications, while keeping complexity $\mathcal{O}(n)$. The reason for using different delays to update the levels of the tree is to leave some time for the changes to settle in the lower levels, which lead to better overall results. Arguably, other data structures could be used instead of binary trees, but binary trees have been used because both their simplicity and straightforward scalability to greater numbers of applications.

## 5.3 Overhead Analysis

This section analyzes the FPCP overhead in terms of hardware and timing complexity for 4- and 8-core systems.

The proposed approach assumes a thread-aware LRU replacement algorithm [11], [14], [48]. Therefore, each cache block is tagged with the associated thread it belongs to (i.e. a 2-bit tag for a cache shared among four threads). As depicted in Table 1, this implies around 35% and 45% of the total overhead for 4 and 8 cores, respectively, which accounts for 1.10% and 1.30% of the area of the entire shared cache (4/8 MB). Notice, however, that some processors already implement this capability; for instance, recent Intel Xeon processors [18], [49] feature cache utilization monitoring and cache partitioning, referred to as Cache Monitoring Technologies (CMT) and Cache Allocation Technologies (CAT), respectively, that can be used for this purpose. Although CAT could be, in principle, used to this end, notice that CAT is designed to be controlled by software, while our approach works at hardware level, with a granularity orders of magnitude smaller (i.e. ns vs ms). Anyway, if the processor implements these features, the total hardware overhead would drop to around 0.71% for both configurations, assuming that we still use the ATD to estimate progress.

The ATD is the other key hardware structure incurring overhead. As mentioned above, this component is used, one per core, to track inter-thread interference. However, to reduce hardware costs we only monitor a subset of 64 cache sets, and the results are extrapolated with minimal impact on accuracy [9]. ATDs account for 64% and 54% of

| Item | General cost | Cost for a 4-core CMP | Cost for a 8-core CMP |
|---|---|---|---|
| Core ID per tag | Cache blocks × $\log_2$ cores | 131072 bits | 393216 bits |
| Alternate Tag Directory | Sampled sets × associativity × (tag + replacement bits) × cores | 237568 bits | 475136 bits |
| Per-core interference cycle counter | 32 bits × cores | 128 bits | 256 bits |
| Counters for the total number of accesses, sampled accesses and inter-thread misses | 3 × 20 bits | 60 bits | 60 bits |
| Per-core counters for hit and miss service times | 16 bits × 2 × cores | 128 bits | 256 bits |
| FPCP tree cost | (2 × cores - 1)× (32 bits + $\log_2$ associativity) | 259 bits | 555 bits |
| Total | | 369215 bits | 869479 bits |
| Percentage area overhead w.r.t. shared cache | | 1.10% | 1.30% |

TABLE 1
Detailed FPCP hardware overhead.

the total overhead, for 4 cores and 8 cores, respectively. The remaining components, included the tree structure required by FPCP, incur in a minimal hardware overhead (less than 1%) and the other values used by the proposal (e.g. CPI and number of cache misses) that do not appear in the table can be gathered from performance counters available in most multi-cores, so they do not incur in additional overhead.

Each $I_1 \times 5$K misses interval, the progress of all the applications is updated in the leaf nodes. Additionally, each $I_i$ intervals, the algorithm listed in Figure 5 (which has a constant execution time) is applied to the Level $i$ of the tree. Therefore, the cost of FPCP is of $\mathcal{O}(n)$ since each interval a number of nodes that depends linearly on the number of applications ($n$) must be traversed. Further timing analysis are discussed in the next section.

## 5.4 Main differences with the ASM-Cache approach

FPCP differs from ASM-Cache both in the criteria applied to distribute cache ways and in the hardware implementation complexity.

ASM-Cache distributes cache ways among applications in a greedy way, according to the estimates of the execution time in isolation. Thus, the number of assigned ways to a given application can highly vary between two successive intervals. For instance, an application could have assigned a few ways (e.g. two out of sixteen) in a given interval and almost all the cache ways (e.g. fourteen or fifteen) in the subsequent interval, or vice versa. As a consequence, inaccurate estimations can severely impact on the system fairness. Unlike this approach, what we propose is to distribute cache space in steps of a single cache way between consecutive intervals. Notice that our proposal is, in essence, based on relative estimates instead of absolute ones. This way makes our approach more resilient to possible inaccuracies in the estimation process.

Regarding complexity, ASM-Cache relies on the cache replacement policy obeying the stack property [13] (as with LRU) and in being able to track the number of hits in each frame of the replacement policy stack. Moreover, since finding the optimal partitioning is a NP-hard problem [12], ASM-Cache employs a $\mathcal{O}(m^2)$ greedy algorithm (where $m$ is the cache associativity) to search for an adequate partitioning. All these reasons difficult the design of a viable hardware implementation. Instead, as discussed above, the cost of FPCP is only $\mathcal{O}(n)$ (with $n$ being the core count).

We experimentally measured the time taken by both the FPCP and other approaches, i.e. ASM-Cache, on a 2.2GHz Xeon, as an approximation of the time taken by the hardware. Regarding FPCP , the algorithm takes between 100 – 800 cycles, depending on the depth of the specific level of the tree being considered. The algorithm is triggered when a given number of cache misses is reached, which translates to around 500K cycles on average, but it is highly dependent on the workload. On the other hand, the time taken by ASM-Cache is about 120K cycles, and the algorithm is triggered each 5M cycles. This means that the overhead of FPCP falls in between 0.02% and 0.16%, while the overhead of ASM-Cache is at least one order of magnitude higher, i.e. by 2.4%. This was expected, since ASM-Cache has quadratic complexity while FPCP ’s complexity is linear.

| | |
|---|---|
| Core count | 2/4/8 cores at 3GHz |
| Issuing policy | Out-of-order |
| Issue/Commit width | 4 instructions/cycle |
| ROB size | 128 entries |
| Load/Store queue | 64/48 entries |
| L1 Icache (private) | 32KB, 8ways, 64B-line, 2cc |
| L1 Dcache (private) | 32KB, 8ways, 64B-line, 2cc |
| LLC (shared) (2 cores) | 2MB, 16ways, 64B-line, 11cc, 16 MSHR |
| LLC (shared) (4 cores) | 4MB, 32ways, 64B-line, 11cc, 16 MSHR |
| LLC (shared) (8 cores) | 8MB, 32ways, 64B-line, 11cc, 16 MSHR |
| Main Memory Latency | 200 cycles zero-load latency |

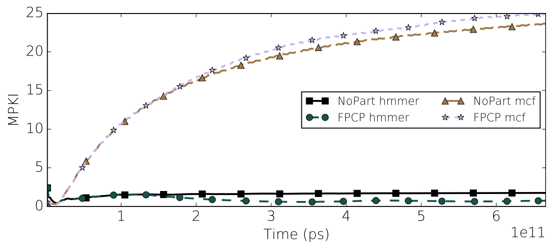
TABLE 2
System configuration.

## 6 Experimental Evaluation

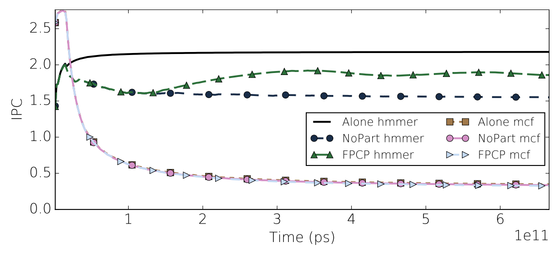
We modeled all the studied approaches and performed a microarchitectural, cycle-by-cycle simulation by extending the Multi2Sim [50] simulation framework. The proposal has been evaluated varying the number of applications (i.e. cores in our system) sharing the cache. We have studied a cache shared by two, four and eight applications, which covers a representative range of shared caches in current multi-cores (e.g. ARM processors).

Each processor core has private 32KB 8-way L1 caches, while the shared cache has a capacity of 1MB per core in the system (e.g. 8MB for the 8-core multi-core). The shared cache for the 2-core processor has 16 ways, while the others have 32 ways. Bank and port contention have been modeled in all the configurations. Table 2 summarizes the main architectural parameters.

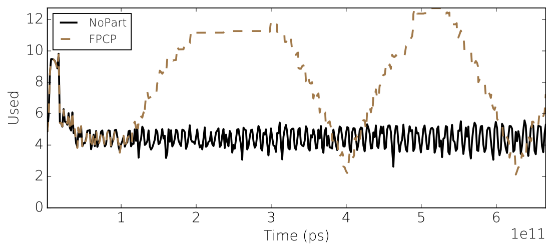
FPCP was evaluated and compared against a baseline shared cache using the LRU replacement policy without partitioning (referred to as NoPart) and against two state-of-the art approaches (i.e. UCP and ASM-Cache). In the UCP scheme, each core has a small utility monitor based



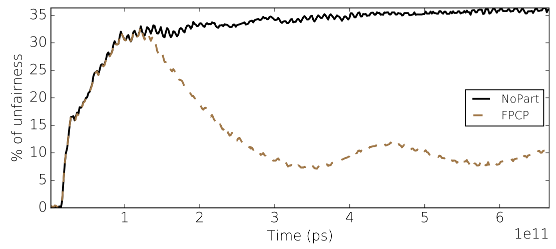
(a) MPKI evolution.



(b) IPC evolution.



(c) Ways used by `hmmer`. The remaining are used by `mcf`.



(d) System unfairness evolution.

Fig. 6. Dynamic `hmmer`-`mcf` evolution.

on dynamic set sampling (UMON-DSS) with 64 sets. This monitor estimates how well each core makes use of cache capacity, and distributes cache resources to minimize the overall number of misses.

With respect to ASM-Cache, it employs a mechanism similar as the used by UCP, but with a different aim. Instead of trying to minimize misses, the mechanism estimates how the slowdown of a given application will be affected according to the number of ways allocated to it, so the optimal partitioning is the one that minimizes the per application slowdown.

Experiments were run with multi-program mixes from both the SPEC CPU2006 benchmark suite [51] with the ref input set and the NAS Parallel Benchmark suite [52] (single-threaded runs). From these benchmark suites three different sets of workloads were considered, varying the number of applications sharing the cache. We used 100 two-application, 175 four-application and 100 eight-application workloads to consider a wide range of scenarios. All the workloads were randomly generated, and results were collected simulating each workload for 2-billion cycles after skipping the initial 500M instructions of each individual application.

## 6.1 Analyzing performance, progress, and unfairness

This section analyzes the dynamic run-time interactions among applications, considering used cache ways, system performance, progress and unfairness. To help the understanding of these interactions and on how our proposal works, we start with a simple two-application example.

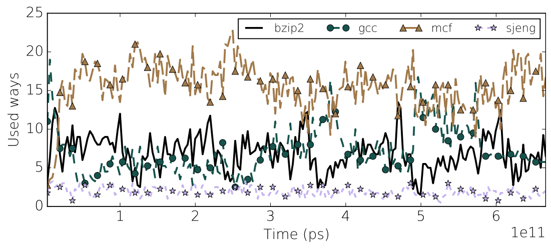
Figure 6 presents the results for two benchmarks, `hmmer` and `mcf`, running concurrently. As we already observed in Figure 1, without cache partitioning this workload exhibits significant unfairness during its execution. This can be also appreciated in Figure 6d, which shows the unfairness evolution for both the baseline approach (NoPart) and for FPCP.

It can be noticed in Figure 6a that the MPKI (Misses Per Kilo-Instruction) at the shared cache of both applications is considerably different; while the MPKI of `mcf` is over 20 for most of its execution, the MPKI of `hmmer` is only around 2. The reason for such a difference is that this fragment of the `hmmer` execution is CPU-bound, and therefore very sensitive to cache misses. This can be appreciated in Figure 6b, where the IPC of `hmmer` drops from above 2 to about 1.6 due to the interference of `mcf`. On the other hand, this phase of `mcf` is memory bound and experiences a high amount of cache misses, so this benchmark shows little sensitiveness to cache miss ratio variations and a cache hog.

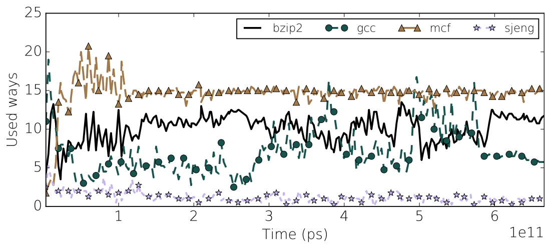
Without any intervention in the shared cache (i.e. without partitioning), this fact translates into noticeable differences in the progress rate experienced by both applications. As shown in Figure 6c, NoPart assigns between 3 and 5 ways to `hmmer` and the remaining cache ways (13 to 11) to `mcf`. Notice that, despite `mcf` holding around two thirds of the cache ways, its IPC is still below 0.5. Moreover, this IPC is similar to what the application achieves in standalone execution. This means that `mcf` exhibits an excellent progress when co-running with `hmmer` (see Figure 1). Therefore, the only way to reduce unfairness is to accelerate the progress of `hmmer` (the least progressing application).

FPCP correctly identifies these progress differences and borrows ways from `mcf`, assigns them to `hmmer`, and improves its progress. As a result, unfairness is reduced from 36% to 10%.

At a first glance, it could seem counterintuitive, since FPCP takes ways from the application with the highest MPKI and assigns them to the one with the lowest MPKI, thus widening even more the huge MPKI differences. However, it makes sense when we realize how little the progress of `mcf` is affected by the number of assigned ways (see Figure 2a). In fact, `mcf` only needs 2 ways to achieve a progress rate of around 90%, while `hmmer` requires around 12 ways to achieve the same progress. This brings two



(a) `NoPart`



(b) `FPCP`

Fig. 7. Run-time way partitioning under NoPart and FPCP for the {`bzip2`, `gcc`, `mcf`, `sjeng`} workload.

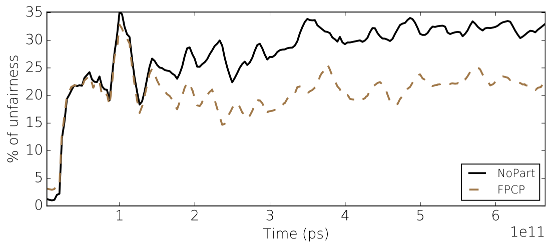


Fig. 8. Unfairness for the {`bzip2`, `gcc`, `mcf`, `sjeng`} workload.

important findings: i) *blindly* reducing cache misses does not necessarily address system fairness, and ii) accurate progress estimations are critical to address fairness.

This analysis can be extrapolated regardless of the number of applications. Lets see another simple working example for a 4-application workload. Figure 7 compares the distribution of ways per application between the non-partitioned baseline approach and the proposal. According to the analysis performed in Section 3, to achieve a progress rate of around 80%, `bzip2` requires about 12 ways, `gcc` 2 ways, `mcf` 8 ways, and `sjeng` 2 ways. However, without cache partitioning, Figure 7a shows that `gcc` occupies at run-time similar or even more ways than `bzip2`, while `mcf` exceeds 16 ways for most of the execution time. FPCP, in contrast, distributes ways much more accordingly (compared to NoPart) to what the progress analysis suggested, giving more ways to `bzip2` and fewer ways to the remaining co-runners, as can be seen in Figure 7b. As a result, FPCP significantly improves system fairness for this workload (see Figure 8), reducing the final unfairness around one third (from 33% with NoPart to 23%).

## 6.2 Experimental results

FPCP has been evaluated against NoPart, UCP and ASM-Cache in terms of system unfairness and performance. Figure 9 presents the unfairness results for 100 pairs of benchmarks sorted in increasing unfairness order (the highest, the worst). Since each application only suffers interference from one co-runner, average system unfairness for this case is relatively low, averaging 7.1% for the baseline (NoPart), 6.2% and 5.7% for UCP and ASM-Cache and 5.6% for FPCP. In spite of this fact, UCP, ASM-Cache and FPCP improve unfairness over LRU by 13%, 19%, and 21%, respectively.

The interference grows with the number of applications running together. Figure 10 shows the results for 175 4-application workloads. Three major observations can be drawn. First, FPCP significantly improves unfairness over NoPart and both state-of-the art approaches. On average, NoPart presents an unfairness level of 28.3%; UCP and ASM-Cache of 21.2% and 20.6%, respectively; and FPCP reduces it to only 13.9%. Second, FPCP highly reduces the maximum unfairness compared to the second best approach (i.e. 37% vs 60% of ASM-Cache). Finally, the standard deviation of the unfairness results achieved with FPCP is much lower compared to the other approaches (20%, 18%, 17% and 10% for NoPart, UCP, ASM-Cache and FPCP, respectively).

To cover a wider range of scenarios, we also evaluate the proposal with 8-application workloads using a shared LLC with 8MB and 32 ways. Again, as shown in Figure 11, FPCP is the approach that keeps unfairness lower, reducing it from the average 50% of NoPart to 36%. UCP and ASM-Cache, while reducing unfairness compared to NoPart in most cases, do not achieve this goal in some specific workloads (as depicted in the right side of the figure). This is because these approaches strongly depend on the accuracy of the estimations, which degrades with the number of applications [9], [10] (see Section 4). Our approach, however, only exchanges one way at a time, so it is more forgiving with progress estimation inaccuracies.

There are multiple reasons why system unfairness rises with the number of cores, even assuming a perfect *progress estimation approach* (i.e. perfect interference estimation). On the one hand, while the per core cache space remains constant (1 MB), the number of ways does not, so the partitioning policy presents comparatively less flexibility. Therefore, under certain circumstances, it cannot provide the high number of cache ways some applications require to have an adequate progress rate. On the other hand, as seen in Figure 2, there are applications whose progress is not affected by the number of cache ways allocated to them (e.g. `CactusADM` or `is`). Thus, when these applications are combined with other applications that require a high number of ways (e.g. `mcf` or `lu`), the system will show an inherently high unfairness level. The reason is that, while the partitioning approaches can equalize the progresses of moderately or highly cache-sensitive applications, the differences between these progresses and the progresses of cache insensitive applications will be important.

Finally, we evaluate the performance of the proposal to verify that fairness is not improved at the cost of performance. Performance results, quantified with the Harmonic Speedup metric [53], [54], show minor differences across the compared approaches. Figure 12 shows the results averaged

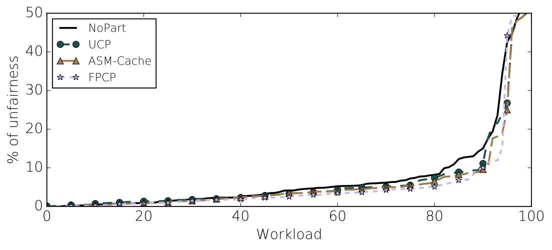


Fig. 9. System unfairness results over 100 2-application workloads.

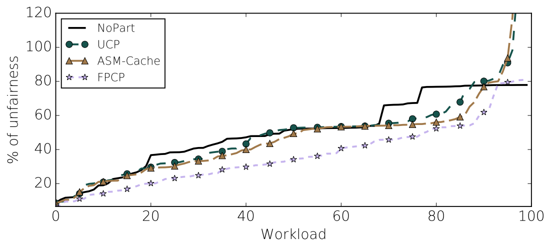


Fig. 11. System unfairness results over 100 8-application workloads.

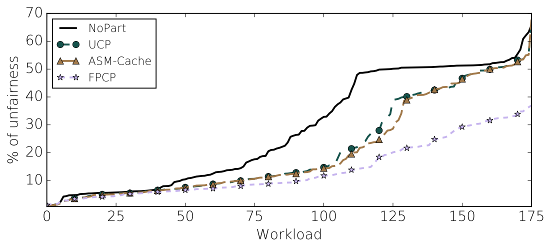


Fig. 10. System unfairness results over 175 4-application workloads.

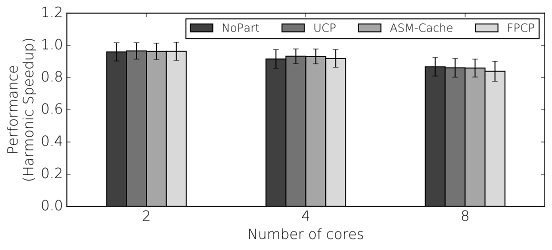


Fig. 12. System performance.

across all the tested workloads for two, four and eight applications.

### 6.2.1 Sensitivity to the aggressiveness of way redistribution

In order to check the unfairness benefits coming from assigning cache ways conservatively either in one-way steps or considering longer intervals, we performed three additional experiments considering 4-application workloads. In the first experiment, we slightly modified FPCP to allow redistributing 2 and 3 ways at a time instead of a single one to check its impact on unfairness. In the other two experiments, we checked the impact of redistributing ways conservatively in other existing approaches. First, we modified the original ASM-Cache to redistribute only one way at a time, and finally, we compared different versions of ASM-Cache by increasing the interval length, thus slowing down way redistribution.

With respect to the first experiment, increasing the number of ways exchanged in each operation, we found that either trading 2 or 3 ways increases unfairness compared to trading only one cache way at a time. This increase is, on average, of around 5%. In the second experiment, we found that limiting to a single one the amount of ways traded each time ASM-Cache is triggered, helped to reduce unfairness. The improvement was, on average, by 6% compared to original ASM-Cache scheme. While this seems significant, note that FPCP has 35% less unfairness, on average, than ASM-Cache. Regarding the third experiment, we compared three versions of ASM-Cache, with 5M-cycle, 10M-cycle and 50M-cycle interval lengths, respectively. While the longest interval slightly reduced unfairness, differences are lower than 3%.

In short, two meaningful findings can be drawn from the aforementioned experiments. First, care must be taken when basing decisions on estimations that can be inaccurate or present transient errors, so slowly redistributing cache ways tends to perform better. And second, conservatively distributing cache ways is not a panacea, but the partitioning algorithm plays a key role, since there are still important unfairness differences between FPCP and the limited ASM-Cache version.

## 7 Conclusions

Cache sharing must be properly managed to address system fairness in current processors.

In this paper, we have characterized the applications in isolated execution and analyzed the relationship between progress and number of assigned cache ways. This analysis gives the progress that each application would achieve in multicore execution if it had assigned during all the execution time a fixed number of cache ways. Thus, it provides some insights about what would be a reasonable *fairness* level to aim for when executing a set of applications concurrently.

A major challenge to address system fairness is the estimation of the execution time of each application is isolation during multicore execution. Two distinct approaches have been evaluated and compared. However, since estimates can be inaccurate and workload behavior fluctuates at run-time, using them directly, as done in the state-of-the-art ASM-Cache and UCP proposals, is not necessarily the optimal approach.

In this paper we have presented FPCP, a simple and effective hardware algorithm that only allows a given application to experience a variation of $\pm 1$ cache way between two consecutive intervals. Compared to ASM-Cache, the complexity of the algorithm is reduced from $\mathcal{O}(n^2)$ to only $\mathcal{O}(m)$, where $n$ is the cache associativity and $m$ the number of cores sharing the cache.

FPCP has been evaluated varying the number of applications running concurrently. Experimental results show that, compared to the state-of-the-art ASM-Cache, FPCP reduces unfairness by 48% in four-application workloads and by

28% in eight-application workloads, without harming the performance.

## Acknowledgments

This work was supported in part by the Spanish Ministerio de Economía y Competitividad (MINECO) and Plan E funds, under grants TIN2014-62246-EXP and TIN2015-66972-C5-1-R.

## References

[1] B. Sinharoy, J. A. V. Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler, "Ibm power8 processor core microarchitecture," *IBM J. of Res. and Dev.*, vol. 59, no. 1, pp. 2:1–2:21, 2015.

[2] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[3] ARM, *ARM Cortex-A53 MPCore Processor. Technical Reference Manual*, 2014.

[4] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi, "Application-to-core mapping policies to reduce memory interference in multi-core systems," in *PACT 2012*, pp. 455–456.

[5] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems," in *ASPLOS*. ACM, 2010, pp. 335–346.

[6] F. J. Cazorla, A. Ramírez, M. Valero, P. M. Knijnenburg, R. Sakellariou, and E. Fernández, "Qos for high-performance smt processors in embedded systems," *IEEE Micro*, vol. 24, no. 4, pp. 24–31, 2004.

[7] R. Gabor, S. Weiss, and A. Mendelson, "Fairness enforcement in switch on event multithreading," *ACM Trans. Archit. Code Optim.*, vol. 4, no. 3, 2007.

[8] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO*. IEEE Computer Society, 2007, pp. 146–160.

[9] K. Du Bois, S. Eyerman, and L. Eeckhout, "Per-thread cycle accounting in multicore processors," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 29:1–29:22, 2013.

[10] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *MICRO*. ACM, 2015, pp. 62–75.

[11] M. Qureshi and Y. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO*, Dec 2006, pp. 423–432.

[12] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A resource allocation model for qos management," in *RTSS*. IEEE Computer Society, 1997, pp. 298–.

[13] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, 1970.

[14] R. Iyer, "CQoS: a framework for enabling QoS in shared caches of CMP platforms," in *ICS*, 2004, pp. 257–266.

[15] J. Chang and G. S. Sohi, "Cooperative cache partitioning for chip multiprocessors," in *ICS*, 2007, pp. 242–252.

[16] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict qos for latency-critical workloads," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 729–742. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541944

[17] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 187–198. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2010.20

[18] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache qos: From concept to reality in the intel® xeon® processor E5-2600 v3 product family," in *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, 2016, pp. 657–668. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2016.7446102

[19] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 450–462. [Online]. Available: http://doi.acm.org/10.1145/2749469.2749475

[20] H. Zhu and M. Erez, "Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 33–47. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872394

[21] L. Funaro, O. A. Ben-Yehuda, and A. Schuster, "Ginseng: Market-driven llc allocation," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '16. Berkeley, CA, USA: USENIX Association, 2016, pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026959.3026987

[22] 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 *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept 2006, pp. 13–22.

[23] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "Addressing fairness in smt multicores with a progress-aware scheduler," in *IPDPS*, 2015, pp. 187–196.

[24] C. Wu, J. Li, D. Xu, P.-C. Yew, J. Li, and Z. Wang, "Fps: A fair-progress process scheduling policy on shared-memory multiprocessors," *TPDS*, vol. 26, no. 2, pp. 444–454, 2015.

[25] D. Xu, C. Wu, P.-C. Yew, J. Li, and Z. Wang, "Providing fairness on shared-memory multiprocessors via process scheduling," in *Performance Evaluation Review*, vol. 40, no. 1. ACM, 2012, pp. 295–306.

[26] A. Fedorova, M. Seltzer, and M. D. Smith, "Improving performance isolation on chip multiprocessors via an operating system scheduler," in *PACT*. IEEE Computer Society, 2007, pp. 25–38.

[27] A. Sharifi, S. Srikantaiah, M. T. Kandemir, and M. J. Irwin, "Courteous cache sharing: being nice to others in capacity management," in *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, 2012, pp. 678–687. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228482

[28] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang, and Y. Luo, "Optimal cache partition-sharing," in *44th International Conference on Parallel Processing, ICPP 2015, Beijing, China, September 1-4, 2015*, 2015, pp. 749–758. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2015.84

[29] S. Kim, D. Chandra, and D. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *PACT*, Sept 2004, pp. 111–122.

[30] H. Dybdahl and P. Stenström, "An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors," in *HPCA*. IEEE, 2007, pp. 2–12.

[31] H. Dybdahl, P. Stenström, and L. Natvig, "A cache-partitioning aware replacement policy for chip multiprocessors," in *HiPC*. Springer, 2006, pp. 22–34.

[32] H. S. Stone, J. Turek, and J. L. Wolf, "Optimal partitioning of cache memory," *Computers, IEEE Transactions on*, vol. 41, no. 9, pp. 1054–1068, 1992.

[33] G. E. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *HPCA*. IEEE, 2002, pp. 117–128.

[34] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *The Journal of Supercomputing*, vol. 28, no. 1, pp. 7–26, 2004.

[35] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *HPCA*, Feb 2009, pp. 250–261.

[36] S. Cho and L. Jin, "Managing distributed, shared l2 caches through

os-level page allocation," in *MICRO*. IEEE Computer Society, 2006, pp. 455–468.

[37] Y. Xie and G. H. Loh, "PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches," in *ISCA*. ACM, 2009, pp. 174–183.

[38] D. Tam, R. Azimi, L. Soares, and M. Stumm, "Managing shared L2 caches on multicore systems in software," in *In Proc. of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2007, pp. 1–8.

[39] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 89–102. [Online]. Available: http://doi.acm.org/10.1145/1519065.1519076

[40] T. Sherwood, B. Calder, and J. Emer, "Reducing cache misses using hardware and software page placement," in *Proceedings of the 13th International Conference on Supercomputing*, ser. ICS '99. New York, NY, USA: ACM, 1999, pp. 155–164. [Online]. Available: http://doi.acm.org/10.1145/305138.305189

[41] Y. Ye, R. West, Z. Cheng, and Y. Li, "Coloris: A dynamic cache partitioning system using page coloring," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Aug 2014, pp. 381–392.

[42] L. Zhang, Y. Liu, R. Wang, and D. Qian, "Lightweight dynamic partitioning for last level cache of multicore processor on real system," in *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, Dec 2012, pp. 33–38.

[43] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *2008 International Symposium on Computer Architecture*, June 2008, pp. 63–74.

[44] M. M. Lee, J. Kim, D. Abts, M. Marty, and J. W. Lee, "Probabilistic distance-based arbitration: Providing equality of service for many-core cmps," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2010, pp. 509–519.

[45] D.-L. Wang, D.-Y. Gao, and D.-H. Wang, "Enhancing the performance and fairness of shared dram systems with sharing-aware scheduling," in *2010 2nd International Conference on Computer Engineering and Technology*, vol. 6, April 2010, pp. V6–709–V6–713.

[46] J. Baer, *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*. Cambridge University Press, 2010.

[47] P. Kaufman, *The new commodity trading systems and methods*. Wiley, 1987.

[48] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *J. Supercomput.*, vol. 28, no. 1, pp. 7–26, 2004.

[49] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual*, September 2015, no. 331843-001US.

[50] R. Ubal *et al.*, "Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors," in *SBAC-PAD*, 2007, pp. 62–68.

[51] *Standard Performance Evaluation Corporation*. [Online]. Available: http://www.spec.org

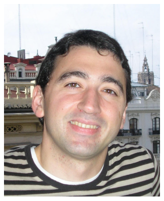
[52] *NAS Parallel Benchmark Suite*. [Online]. Available: http://www.nas.nasa.gov/publications/npb.html

[53] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, May 2008.

[54] K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in smt processors." in *IsPASS*, vol. 1, 2001, pp. 164–171.

**Vicent Selfa** received his BS and MS in Computer Engineering from the UPV, Spain in 2013 and 2014, respectively. He is currently a PhD student at the Parallel Architecture Group (GAP) of the Universitat Politècnica de València with a fellowship from the Spanish Government. His research interests include fairness-aware resource partitioning policies and chip multiprocessor architectures.

**Julio Sahuquillo** received his BS, MS, and PhD degrees in Computer Engineering from the UPV, Spain. Since 2002 he is an Associate Professor at the DISCA department at the UPV. He has published more than 140 refereed conference and journal papers. His current research topics include multi- and manycore processors, memory hierarchy design, and architecture-aware scheduling. He has cochaired several workshops, collocated in conjunction with IEEE supported conferences.



**Salvador Petit** received his PhD degree in computer engineering from the UPV. Currently, he is an associate professor in the DISCA department at UPV where he teaches several courses on computer organization. His research topics include multithreaded and multi-core processors, memory hierarchy design, as well as real-time systems. He is a member of the IEEE and the IEEE Computer Society.

**María E. Gómez** received his BS, MS, and PhD degrees in Computer Engineering from the UPV, Spain. She joined the DISCA department at UPV in 1996 where she is currently an Associate Professor. She has published more than 50 conference and journal papers. She has served on program committees for several major conferences. Her research interests are on processor architecture and interconnection networks.