Application Clustering Policies to Address System Fairness with Intel's Cache Allocation Technology

Vicent Selfa*, Julio Sahuquillo*, Lieven Eeckhout†, Salvador Petit* and María E. Gómez*

* Dept. of Computer Engineering
Universitat Politècnica de València, Spain

† Dept. of Electronics and Information Systems
Ghent University, Belgium

***Abstract*—Achieving system fairness is a major design concern in current multicore processors. Unfairness arises due to contention in the shared resources of the system, such as the LLC and main memory. To address this problem, many research works have proposed novel cache partitioning policies aimed at addressing system fairness without harming performance. Unfortunately, existing proposals targeting fairness require extra hardware which makes them impractical in commercial processors.**

Recent Intel Xeon processors feature Cache Allocation Technology (CAT), a hardware cache partitioning mechanism that can be controlled from userspace software and that allows to create partitions in the LLC and assign different groups of applications to them.

In this paper we propose a family of clustering-based cache partitioning policies to address fairness in systems that feature Intel's CAT. The proposal acts at two levels: applications showing similar amount of core stalls due to LLC accesses are first grouped into clusters, after which each cluster is given a number of ways using a simple mathematical model. To the best of our knowledge, this is the first attempt to address system fairness using the cache partitioning hardware in a real product. Results show that our best performing policy reduces system unfairness by up to 80% (39% on average) for 8-application workloads and by up to 45% (25% on average) for 12-application workloads compared to a non-partitioning approach.

I. Introduction

Current multicore processors typically implement huge last-level caches (LLC) to hide the large main memory access latencies. The size of these caches ranges from several tens of MBs to hundreds of MBs in recent processors like the IBM Power8 or the Intel Xeon Phi Knights Landing. Because of their large storage capabilities, as well as their high associativity (e.g., more than 16 ways), these caches are typically shared among all the cores in the processor. By default, all the running applications compete among each other for LLC space, which is governed by a single replacement policy. As a consequence, the applications replace blocks that belong to other applications, which can seriously degrade their performance. Moreover, it is difficult to predict the effect of these inter-application interactions, since depending on their characteristics, some applications are affected more than others. That is, while the performance of some applications can be highly degraded, other applications may be unaffected, creating a fairness problem in the system.

Much research has focused on cache sharing over the past decade. Some of these works concentrate on performance [2], [26], [34], [37]; others target LLC cache fairness [9]; and yet others focus on providing system fairness [14], [43], [45]. The latter works consider system components other than the LLC (e.g., the memory controller). The vast majority of these works, however, present four main drawbacks that render their conclusions either invalid or inapplicable to recent processor generations. First, most of these works consider the L2 cache as the LLC, mainly because their research is performed in simulators, in which filling up a huge LLC of tens of MBs would require a prohibitive amount of simulation time. That means that since neither the cache geometry nor the data locality match, results cannot be easily extrapolated to recent commercial machines. Second, most of these works do not take into account the impact of hardware prefetchers or do not model the prefetchers employed in commercial machines, often not well documented. Third, these works either do not model the memory controller or model a simplified version. Fourth, some of these approaches require the use of extra hardware to obtain their inputs (e.g., the number of cache misses specifically caused by other co-runners). Since any runtime approach that deals with fairness has to estimate the slowdown the applications are suffering, most of previous research targeting fairness has this problem. Notable examples are the Per-Thread Cycle Accounting Architecture [7], [12] and the Application Slowdown Model [41]. Both approaches require extra hardware that is not readily available in any commercial processor.

The results of the discussed research regarding cache partitioning, however, were so promising that some processor manufacturers have implemented cache partitioning capabilities in their products. This is the case for recent Intel processors that feature the Cache Allocation Technology (CAT), which provides primitives to limit the amount of cache space a hardware thread can occupy in the LLC.

More precisely, CAT allows for a given number of ways to be assigned to a specific set of processes, a *Class of Service* or CLOS in Intel terminology. As there can be much more processes than classes of service, processes must be mapped to classes following a given policy. Therefore, a policy providing a limited number of cache ways to each application as done in previous works [34], [41] is unsuitable, since it would require a different CLOS for each application. While this problem could be solved by assigning multiple applications to the same CLOS, we have characterized the slowdown each application experiences over isolated execution varying the number of LLC ways, and the results of this study show that assigning an exclusive subset of ways to each CLOS significantly reduces both system throughput and fairness compared to allowing different classes of service to share LLC ways.

This work proposes a family of clustering based cache partitioning policies that leverage the capabilities of Intel's Cache Allocation Technology to deal with system fairness. Although the proposal has been evaluated on an Intel Xeon E5 2658A v3, it is straightforward to port to any processor supporting CAT. It works by applying clustering techniques to group applications suffering from similar core stall cycles due to L2 misses into the same CLOS, and giving each CLOS an adequate number of LLC ways.

In this paper we make the following key contributions:

- We propose a family of cache partitioning policies based on application clustering to improve system fairness on recent real machines.
- To the best of our knowledge, our proposal is the first to leverage state-of-the-art cache partitioning technologies, i.e., Intel's Cache Allocation Technology (CAT), to improve system fairness.
- We comprehensively evaluate the devised policies against the original system with no cache partitioning, and demonstrate improvements in system fairness by up to 80% (39% on average) for 8-application workloads and by up to 45% (25% on average) for 12-application workloads for a range of multiprogram workloads on modern hardware. This is done without significantly affecting the performance for 8-application workloads and improving it for 12-application workloads.

II. Progress, Slowdown and Unfairness

Before diving into the specifics of Intel's CAT, we first introduce a number of important performance metrics. This paper makes extensive use of the metrics *progress*, *slowdown* and *unfairness* for evaluation purposes. The progress of an application is computed [11], [13], [16], [43] as the ratio of its execution time while running with other applications, relative to its execution time in isolation[1]:

$$Progress = \frac{ExecCycles_{alone}}{ExecCycles} \tag{1}$$

Progress is, therefore, a value between 0 and 1. Slowdown is the inverse of progress; thus, it is a value always equal to or larger than 1. Both metrics are used to measure how interference between applications degrades performance. When progress and slowdown are equal to 1, the performance of the application is not affected by the other competing applications.

After measuring the interference with these metrics, we can define a system as completely fair when all the tasks in the system experience the same progress or slowdown [2], [9], [16], [33].

Based on this definition, some works [6] propose the ratio between the progress of the application that progresses the most and the application that progresses the least as a way to quantify unfairness. However, this metric only considers extreme values, so to overcome this limitation in this work we employ an alternative metric proposed in [43], which uses the coefficient of variation (CoV) [10] of the slowdowns of the running applications with respect to the mean to measure how unfair the system is, as shown in Equation 2, where σ refers to the standard deviation and μ to the mean slowdown:

$$Unfairness = \frac{\sigma_{Slowdown}}{\mu_{Slowdown}} \tag{2}$$

For measuring system performance, we use System Throughput (STP, Equation 3) and Average Normalized Turnaround Time (ANTT, Equation 4), two well-defined metrics that are extensively used in the literature [11], [13], which compute the aggregated progress and the average slowdown, respectively:

$$STP = \sum_{t \in Tasks} Progress_t \tag{3}$$

$$ANTT = \mu_{Slowdown} \tag{4}$$

Both metrics quantify different aspects of multiprogram performance (overall system performance versus per-application performance), so when evaluating a system we need to consider both.

III. Cache Allocation Technology

This work leverages Intel's Cache Allocation Technology (CAT) to improve system fairness. CAT provides primitives to limit the amount of LLC space a hardware thread can occupy. It is available in a limited set of processors of the Xeon E5 2600 v3 family and all the processors of the Xeon E5 v4 family.

Machines with Intel's CAT have a predefined amount of classes of service (CLOS), 4 in Haswell EP machines like those in the Intel's Xeon E5 2600 v3 family and 16 in Broadwell EP machines from the Xeon E5 2600 v4 family. Each CLOS has a *capacity bitmask* (CBM) that controls the accessibility of cache resources with cache way granularity, where each bit in the mask grants write access to one way in the cache. Additionally, a CLOS has a list of thread IDs that belong to the CLOS, which are the ones that have write access to the ways set in the CBM. CBMs can overlap, which means that some ways can be shared by different classes of service. One of the main limitations of CAT is that all the

[1]Note that progress is considered on a per-application basis, regardless of the application type, i.e., single-threaded or multi-threaded.

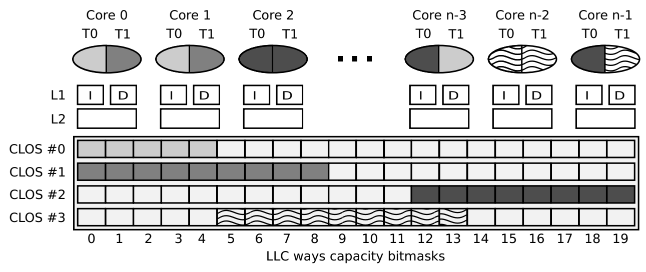

Fig. 1. Cache Allocation Technology example.

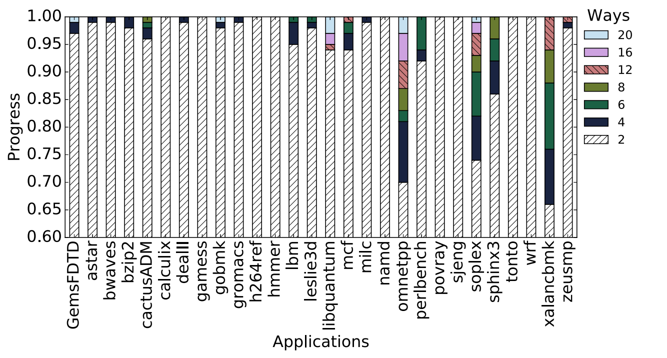

Fig. 2. Effect of the number of cache ways on progress.

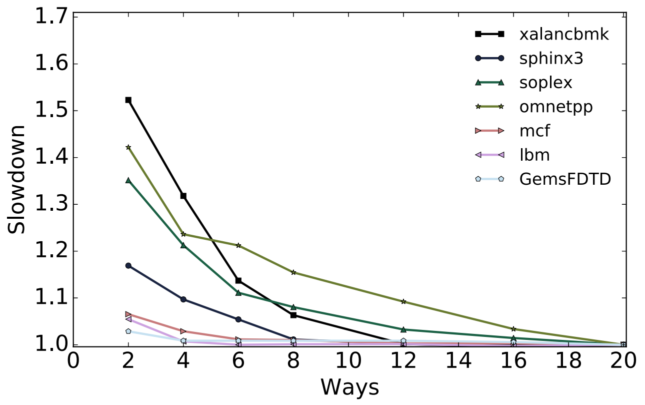

Fig. 3. Slowdown when varying the available ways with respect to a 20-way cache.

bits set in a CBM must be consecutive. That is, a CLOS uses consecutive cache ways in the cache. For instance, a CBM like `1111-0000-1111-0000` would not be valid.

One can configure CAT by directly writing to machine specific registers (MSR), but a better option is to use a library developed by Intel [23] or the interface provided by the Linux kernel starting from version 4.10. By default, CAT has no effect, since all applications are mapped to CLOS #0, which, if not modified, has a CBM that allows full access to all the LLC ways.

Figure 1 shows an example of a possible cache partitioning scheme in a processor of the Xeon E5 2600 v3 family. Each of the four possible classes of service (CLOS #0 – CLOS #3) has assigned a subset of the 20 ways of the LLC, and each thread is assigned to a CLOS. Each CLOS is identified by a color/pattern which marks both the threads that belong to the CLOS and the ways they can write. For instance, thread 0 of core $n-3$ is assigned to CLOS #2 and thread 1 to CLOS #0. Note that all the CBMs are contiguous and that CLOS #3 shares some of its assigned ways with CLOS #1 and CLOS #2.

IV. Progress Characterization and Estimation

The main aim of the cache partitioning scheme proposed in this paper is to balance the progress among applications to improve system fairness. In concurrent execution, the cache interference caused by other applications reduces the *effective* number of cache ways a given application can use. To explore the sensitiveness of individual applications to the number of available cache ways, we conduct several experiments in which we use CAT to adjust the number of ways available to the application from 2 to 20 (i.e., the total cache space). Using CAT, we prevent the application under study from using non-allocated ways. This approach allows modeling the reduction in the available cache space for a given application due to the cache interference induced by co-runners. This provides a reproducible way to study how progress is affected by co-runners competing for cache space.

Figure 2 shows the progress results for different applications of the SPEC CPU2006 benchmark suite. As observed, for a given number of assigned cache ways, applications achieve different progress levels. That is, the progress of each application exhibits a distinct sensitiveness to the available cache space. There are *highly cache sensitive* applications, like `xalancbmk`, `soplex` or `omnetpp`, whose progress is significantly harmed when the number of assigned ways drops below 6. In contrast, some applications are not affected at all; that is, they achieve 100% progress with only 2 cache ways. The rest of the applications fall somewhere in between, with different degrees of cache space sensitiveness. Note that each way represents 1.5MB of the total cache space (i.e., 30MB), so two ways are equivalent to 3 MB of LLC cache space, which is already a considerable amount.

Another way to look at this issue is to analyze how the number of assigned ways affects the slowdown. For this purpose, we plot the slowdown of each application as a function of the number of assigned ways. Figure 3 illustrates the results for some applications of the SPEC suite. Looking at *highly cache sensitive* applications like `xalancbmk`, we find that the slowdown when varying the cache space can be modeled using an exponential function $a \cdot e^{-x} + b$, where a and b are constants that depend on the application and x is the cache space. We also explored other approximations, including a linear, quadratic and cubic function, however, we find the exponential function to yield the best fit. For instance, `soplex` follows the equation $5.24 \cdot e^{-x} + 0.026$.

The previous finding suggests that the slowdown grows exponentially as the number of assigned cache ways is re-

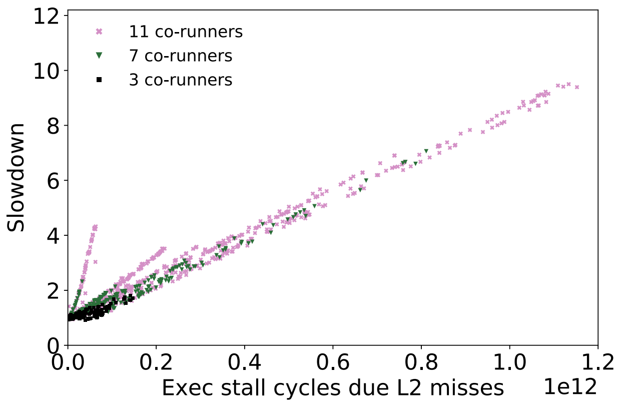

Fig. 4. Correlation between L2 miss stall cycles and slowdown on the Intel Xeon E5 2658A v3.

duced. Or, inversely, for having a linear reduction in slowdown we need an exponential increase in cache space. Note that previous research has found a square root relationship between cache space and hit ratio [4], [19].

Theoretically, we could use these equations to determine the assignment of cache ways. However, the *real* slowdown that a given application is suffering at runtime cannot be directly calculated since the execution time of the applications in isolation is unknown. Several previous works have focused on estimating this execution time. However, most of them require additional hardware [7], [9], [12], [41] to calculate the number of cycles the processor is stalled due to interference in the shared resources, or need to modify the OS scheduler [14], [45].

Our work targets unmodified commercial processors running a vanilla Linux kernel, so since we could not use previous research, we looked into the available performance events related to processor stalls due to shared resources, and studied the correlation between them and overall application slowdown. We find that the $STALLS\_L2\_PENDING$ performance counter is the one that best correlates with application slowdown. This counter gathers the number of cycles during which the execution of an application is stalled due to L2 cache misses. Figure 4 plots slowdown versus the number of stalls (in trillions) gathered by the mentioned performance counter for several runs of SPEC applications executed with different numbers of co-runners (3, 7 and 11 random co-runners). As observed, there is a strong positive correlation (r=0.982, N=833, p=0.000) between the slowdown metric and the $STALLS\_L2\_PENDING$ count.

This correlation can be explained with the following rationale. The $STALLS\_L2\_PENDING$ counter is affected by the interference in all the shared resources in our experimental platform, which are the LLC, the main memory, and the on-chip interconnects (two rings connecting the L3 slices and the memory controllers) as depicted in Figure 5. This performance counter does not differentiate between stall cycles caused by *normal* misses and interference misses, but as the number

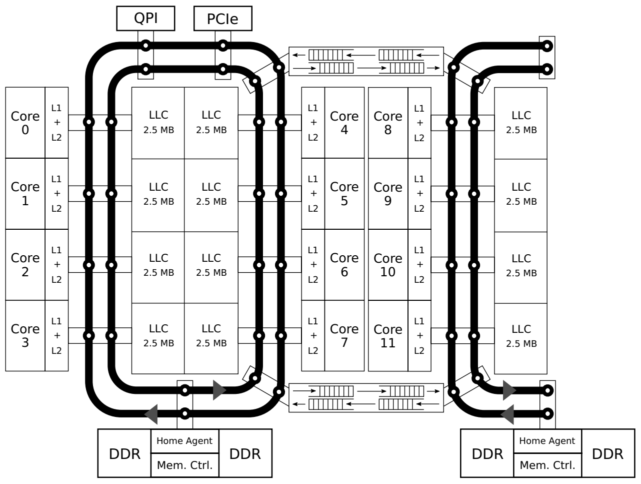

Fig. 5. Block diagram for the Intel Xeon E5 2658A v3.

of concurrently running applications grows, the stall cycles due to interference start to dominate (the ANTT for 8 and 12 concurrently running applications is, on average, over 3 and 4, respectively) so this drawback becomes less important.

Another way to understand this correlation is to examine the following equation:

$$Slowdown = \frac{ExecCycles}{ExecCycles_{alone}} =$$

$$= \frac{CoreCycles + STALLS\_L2\_PENDING}{ExecCycles_{alone}}, \quad (5)$$

where $CoreCycles$ represents the execution time minus the cycles stalled due to L2 misses. For high enough slowdowns, the result of this equation is dominated by STALLS_L2_PENDING.

Finally, as observed in Figure 4, there are some points that deviate from the main trend. These deviations are due to L3 block replacements that cause L2 invalidations to keep the inclusion principle (notice that the L2 cache is private and the whole cache hierarchy is inclusive [24]). In turn, these invalidations produce additional L2 misses that increase the number of $CoreCycles$. We verified this hypothesis by analyzing the data used for Figure 2 in which we varied the number of available ways for each application. As expected, we find that the applications that present a deviant behavior in Figure 4 have a sudden increase in L2 evictions when the cache space is reduced (2 or 3 ways in the LLC). This only happens for the applications that exhibit such behavior and not the rest.

This effect could be taken into account using a different performance event that considers L2. For instance, there is a performance counter that gathers the number of execution stalls due to L1 misses (*STALLS_L1D_PENDING*). Unfortunately, this counter does not behave correctly in our experimental platform (its value is always zero).

V. To Overlap or Not To Overlap Cache Ways

Most previous cache partitioning approaches work by assigning cache ways to applications to be used *exclusively*. That is, a given cache way can be only used by one application.

When using CAT capabilities, however, a wider design space opens. In addition to assigning ways exclusively to individual applications, cache ways can be: (i) allocated to a single CLOS hosting a set of applications (i.e., limited sharing), and (ii) allocated to multiple classes of service. To the best of our knowledge, only the first design choice has been considered in previous research [15], [31], [49]. Moreover, unlike this work which targets fairness, the focus of these approaches is on Quality-of-Service, and CAT capabilities are used to isolate latency-critical applications.

Assigning all the ways as private to individual applications becomes quite unrealistic on real hardware using CAT, mainly due to the high number of potentially running applications, the limited number of supported classes of service, and the limited number of cache ways. For instance, the Intel Xeon E5 2600 v3 family has 20 ways in the LLC and supports 4 CLOS. Therefore, no more than four applications could have exclusive ways assigned.

Even in the most recent Intel Broadwell EP machines that support up to 16 classes of service, the number of concurrently running applications can be higher, and the number of ways in the LLC is still limited to 20. Thus, if a workload with eight applications is executed, each application would have on average 2.5 ways, which clearly is insufficient to reach reasonable performance for most of them. Other partitioning schemes could be tried, giving more ways to some applications and less to others, but since there are applications that require a high number of ways to perform well (see `xalancbmk`, `omnetpp` and similar applications in Figure 2) this approach does not scale well and cannot be generalized.

Although grouping applications in classes of services that can access disjoint sets of ways may seem like a viable solution, it has the same problems as the previous approach, because while it partly solves the limitation in available classes of service, the number of ways in a CLOS is still too small for some applications to meet reasonable performance goals.

Two experiments were carried out to verify this claim. In one, the cache is divided in partitions of the same size for each CLOS and we try different clustering schemes to map 8 applications to 4 classes of service. In the other, we try several partitioning schemes with each partition being of a different size. In each case we also try different approaches to group applications (e.g., KMeans clustering, complementary cache requirements, similar cache requirements, etc.). In both experiments, throughput is significantly degraded without improving fairness with respect to no partitioning.

Consequently, all the partitioning approaches proposed in this work allow for overlapping LLC cache ways (i.e., CBMs) among classes of service.

VI. Cluster-Based Partitioning Policies

As mentioned in Section IV, applications running on a multicore processor suffer from slowdown due to interference that arises from resource sharing. The interference varies during execution time depending on the run-time resource requirements of the applications and, since not all the applications have the same requirements and are not affected equally, unfairness arises. Our goal is to smartly partition cache resources to counteract the slowdown inequalities, which leads to a fairer system.

A cache partitioning mechanism can be characterized by three main design aspects [20]: target, evaluation metric and policy metric. Our proposal targets fairness, and the evaluation metric we use is the one defined in Section II, the CoV of the slowdowns. This metric cannot be computed online, because the execution time alone cannot be measured at run-time, so using the insights presented in Section IV, we use the CoV of the $STALLS\_L2\_PENDING$ as the metric that guides our policies. Specifically, the goal is the minimize the CoV of the $STALLS\_L2\_PENDING$ counter.

Additionally, when using CAT for partitioning the cache three main design decisions must be taken: (i) the number of partitions in the LLC, (ii) which applications are assigned to each partition, and (iii) the amount of resources assigned to each partition.

Each design decision can, in turn, be either statically established or dynamically adjusted at run-time. These three axes open a new design space, summarized in Table I, which greatly affects the performance and fairness that a policy provides. In this work we explore it and present the most relevant results. Although policies for each type that made sense have been devised and evaluated, for the sake of clarity only results for the best performing ones are presented in this paper.

Taking all of this into account, we propose a family of application clustering algorithms, based on the $STALLS\_L2\_PENDING$ event. They target fairness and cover the key issues of the design space. The family consists of three main policies, namely SFn-mK, mK, and Dunn, where n and m are parameters of the policies, whose meaning is described below. The differences between policies mainly arise due to two aspects: the number of clusters the policy builds at run-time, and the form in which cache ways are assigned to clusters (i.e., fixed or dynamic).

All the proposed policies group applications in clusters using the KMeans algorithm [18] according to the number of core stalls due to L2 misses. Given n one-dimensional data points (only one variable — core stalls — is being considered per application), this algorithm distributes them into k clusters, assigning each application to its closest cluster, where the *closeness* is calculated as the Euclidean distance between the data point and the cluster centroid. A major advantage of using one-dimensional data (as in this case) is that an optimized version of KMeans can be used, with $\mathcal{O}(k \cdot n \cdot \log n)$ complexity. Once the clusters have been obtained, all the applications in a given cluster are assigned to the same CLOS.

TABLE I
DESIGN SPACE AND EVALUATED POLICIES. LEGEND: S = STATIC AND D = DYNAMIC.

Num. Clusters	Cluster sizes	Ways per cluster	Policies
S	S	S	
S	S	D	
S	D	S	SFn-mK
S	D	D	mK
D	S	S	
D	S	D	
D	D	S	
D	D	D	Dunn

The number of LLC ways assigned to each CLOS and the number of clusters used depend on the specific policy.

SFn-mK Policies. In these policies, applications are grouped in m clusters using the KMeans algorithm. After the clustering process is done, the m clusters are sorted according to their centroid values in descending order and mapped to different classes of service. The cluster whose applications are suffering the highest slowdown (i.e., the most critical one) is given the highest priority and it is allowed write access to all the cache ways (i.e., its CBM is set to 0xFFFFF). The following clusters receive a decreasing number of ways in steps of n according to their criticality. For example, for $n = 3$ and $m = 4$, the four clusters, sorted in critical order, receive 20, 17, 14 and 11 cache ways, respectively. Different values of n and m, ranging from 2 to 4, have been evaluated. In this work, we only show results for the policies of the form SFn-4K, which were the best performing.

mK Policies. These policies also group applications using the same criterion as the previous group of policies, but the number of ways assigned to each partition is not static but computed using a simple exponential function. We chose an exponential function because looking at Figure 3, to have a linear reduction in slowdown an exponential increase in cache space is required. Other functions were explored, such as linear and quadratic functions, but the exponential approach was the one providing the best results. The input to the exponential function is the normalized stalls of each cluster with respect to the most critical one, a value in the interval [0..1]. The output of the function is the number of assigned cache ways for the cluster, a value in the interval [2..20]. Figure 6 depicts the behavior of this policy for m clusters. In the example of the figure, cluster $m - 1$ is the most critical one, and the number of assigned ways is 20, 10, 4, and 2, for the clusters $m - 1$, $m - 2$, 1, and 0, respectively.

Dunn Policy. This policy follows the same approach as the mK policy regarding clustering and the assignment of cache ways to classes of service. The number of classes of service to use is, unlike previous policies that consider a fixed number of clusters, dynamically determined at runtime to adapt to the different phases of the workload execution. To this end, two indices to evaluate clustering validity and determine the optimal number of clusters (Silhouette [35] and Dunn [8]) have

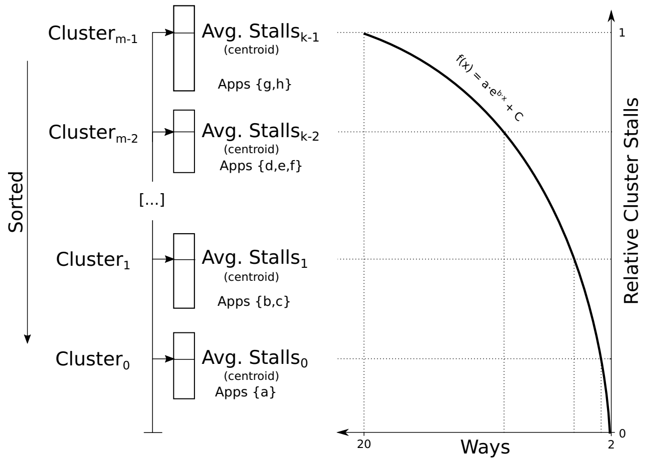

Fig. 6. Policy with clustering and a model for the ways.

been evaluated. In this paper, results are only shown for the policy using the Dunn index, since it yielded slightly better results on our experimental platform, thus we refer to this approach as the Dunn policy.

The Dunn index is defined as follows. Assuming k denotes the number of clusters, d_{min} the minimal distance between points of different clusters, and d_{max} the largest within-cluster distance; the Dunn index for k clusters is then computed as

$$Dunn_k = \frac{d_{min}}{d_{max}}.$$

The closest the Dunn index to 0 the better. As a result, the k value that minimizes the Dunn index is selected.

VII. EXPERIMENTAL SETUP

All the experiments have been performed on the Intel Xeon E5 2658 v3 processor [21], that has been one of the first Intel processors to support Cache Allocation Technology.

This processor implements HyperThreading and is deployed with twelve cores supporting up to two simultaneous threads each. However, to avoid intra-core interference, experiments have been conducted allocating a single thread per core. Each core includes a 32 KB L1 data cache and a 256 KB L2 cache, both of which are private. All the cores share an L3 cache, the LLC in the system, with 30 MB and 20 ways (which gives an average of 2.5 MB per core). The entire cache hierarchy is inclusive [24].

The proposal focuses on the LLC, where the space is distributed according to the different devised policies using CAT. In order to make the experiments repeatable, important system details are provided next. The processor frequency was fixed to 2.20 GHz with the Linux 4.9.0 performance governor. The main memory of the system has a maximum theoretical bandwidth of 68 GB/s across 4 channels. We experimentally measured memory latencies of 75 ns for an idle machine and 570 ns for a saturated machine. The machine has 4 types of hardware prefetchers: 2 prefetchers associated with the L1-data cache and 2 prefetchers associated with the L2 cache. All of them were kept enabled during the experiments.

All the devised schemes, explained in Section VI, have been evaluated and compared against a baseline that performs no partitioning (referred as NoPart).

The experiments have been conducted with two sets of 45 multiprogram mixes from the SPEC CPU2006 benchmark suite [1] using the reference input set. The first set contains 8-application workloads and the second 12-application workloads. To compose the application mixes, we first classify the applications in the SPEC benchmark suite into two categories, cache sensitive and cache insensitive, based on the offline evaluation performed in Section IV. Then, we create workloads varying the ratio of sensitive to insensitive applications. All the workloads are randomly generated, and results are collected executing each workload until all the applications have completed the same number of instructions they execute when running alone on the machine for 60 seconds.

Our experimental environment consists of a *manager* program that reads a configuration file with a list of workloads and the partitioning policy that will be used. The *manager* then *forks* and *execs* as many times as necessary to launch the applications in the workload. At regular intervals of 500 ms the *manager* reads the required performance counters and uses this information to properly size the partitions and assign applications to classes of service. We tried other two different intervals for adjusting the partitioning: 100ms and 1000ms. In the first case there was no significant difference in the results, but the overhead was higher, since the manager was active more frequently. In the second case the results were worse compared to the same configuration with a smaller interval.

When an application executes as many instructions as it would run in isolation during 60 seconds, the *manager* restarts it. However, only the results from the first run of each application are taken into account. Note that due to limitations in the libraries employed for performance monitoring [22] and cache partitioning [23], applications need to be pinned to cores[2].

Each experiment has been repeated *a minimum* of 10 times, until the margin of error was less than 1%, with a confidence of 95%. This applies to all the plots and data shown in this paper, so no confidence intervals are drawn on the figures.

VIII. Evaluation

This section is aimed at providing insights and quantify the benefits of the devised policies. To this end, the evaluation focuses on three main design concerns: (i) Can a simple static policy provide significant fairness? (ii) Should the number of clusters match the maximum number of CLOSes supported by the machine all the time? (iii) Would dynamically adapting the number of classes of service to the *optimal* number of clusters further improve the results? The three devised policies and the experiments discussed below were designed to answer these three questions.

[2]Starting from version 4.10, the Linux kernel has native support for CAT, and it does not have this limitation. Unfortunately, it was not available when the experiments were performed.

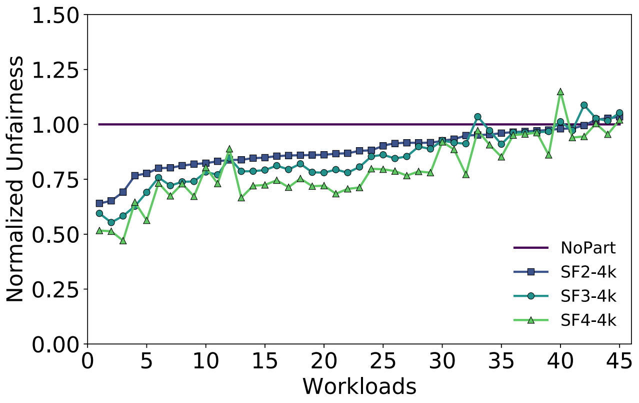

Fig. 7. Normalized unfairness across the 45 8-application workloads with the SF2-4K, SF3-4K and SF4-4K policies.

A. Exploring unfairness enhancements with a simple policy (static number of clusters, static number of ways)

To explore the potential of CAT to help facing unfairness, we evaluate the simple SFn-4K policy across the studied 45 8-application workloads. Remember that this policy assumes four clusters, and the number of ways assigned to each cluster is decreased from 20 (assigned to the highest priority cluster) in steps of n. In this experiment, we vary the value of n from 2 to 4.

Figure 7 shows the results. To plot the curves, we first have sorted the 45 workloads in ascending order depending on the normalized unfairness they present using the SF2-4K policy. Then, SF2-4K and SF3-4K have been plotted following the same order. As observed, this policy, by merely assigning a static number of ways to each CLOS and using 4 classes of service improves unfairness significantly, by 12%, 16% and 21% on average for SF2-4K, SF3-4K and SF4-4K, respectively. Also note that not all the mixes obtain the best results with the same value of n. Although SF4-4K presents the best results, in around 10% of the workloads, another value of n yields better results.

Important conclusions can be drawn from this experiment. First, the $STALL\_L2\_PENDING$ counter acts as a good criterion to group applications in clusters. Second, regardless of the value of n, unfairness is significantly improved over the non-partitioning approach. Third, CAT presents high potential to improve system fairness even with a simple policy based on clustering.

B. Determining the number of cache ways assigned to each cluster dynamically

Section VIII-A, using a simple policy and exploring only three values of n, has shown that there is not a number of ways that can be assigned statically to clusters to provide the best results for every workload. In other words, there is not a single optimal n value for all the workloads. Additionally,

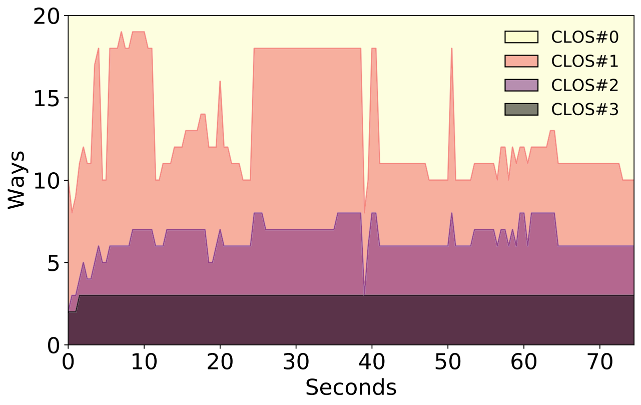

Fig. 8. Ways assigned to each CLOS during the execution of the 4th 8-application workload.

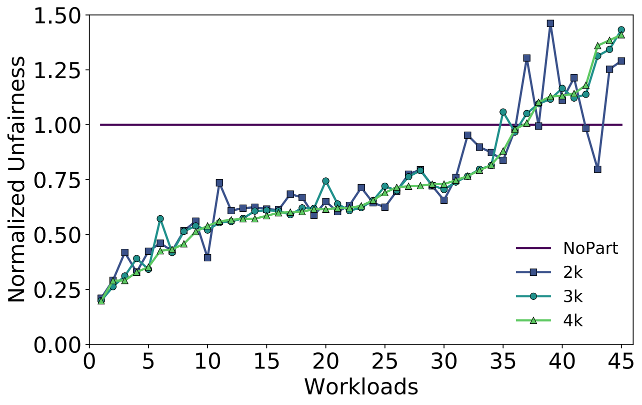

Fig. 9. Normalized unfairness across the 45 8-application workloads with the 2K, 3K and 4K policies.

although not measured in the previous experiment, cache space requirements of the applications in a workload can change during the execution so being able to adapt to these changes can provide further unfairness improvements.

To deal with the shortcomings of a static way allocation, the mK policy was devised, which dynamically assigns the number of ways that best fits the cache space requirements for each of the application clusters. This policy allows greater flexibility, since the number of ways assigned to each cluster is adjusted dynamically at runtime. Figure 8 illustrates how the number of ways assigned to each CLOS varies during the execution of the 4th 8-application workload running under the 4K policy (i.e., grouping applications into 4 clusters). As observed, unlike the previous policies (the SFn-mK group of policies), the number of cache ways assigned to each cluster varies at run-time, which allows this policy to bring important benefits over the SFn-4K, as our results will show.

The mK policy has been evaluated while setting the value of m to 2, 3 and 4 clusters, see Figure 9. The 45 8-application workloads have been sorted in increasing unfairness order, according to the results of the 4K policy. The 2K and 3K policies have been plotted following the same order. Counter-intuitively, not always the maximum number of clusters (i.e., 4) shows the best results; even more, on average, the policy with two clusters (2K) yields slightly better results than the one with 4 clusters (4K). The reason is that changing the clustering (i.e., the number of clusters and the applications in them) displaces the centroids and thus can have a significant impact on the number of assigned cache ways to each cluster. Moreover, less clusters oftentimes means more ways per CLOS, which improves the performance of applications with high cache space requirements.

Taking into account these results and comparing them with the ones presented in Figure 7, two important conclusions can be drawn. First, the major fairness benefits come from the fact that cache ways are dynamically assigned to clusters at run-time, rather than the number of clusters itself. Notice that in Figure 9, the normalized unfairness starts from below 0.25 while in Figure 7 it starts from around 0.5. Second, additional benefits could be brought by dynamically selecting the proper number of clusters.

Note that some workloads experience an increase in unfairness (see rightmost data points in Figure 9). This is the case for workloads composed mostly of applications that are not cache-sensitive, so the absolute (non-normalized) unfairness is low. Their absolute unfairness is around 0.07, which is significantly less than the global average around 0.23. So, while in some corner cases we see a small increase in unfairness in absolute terms, this is compensated for by large reductions in unfairness (both in absolute and relative terms) in the general case. The explanation for this behavior is that in low unfairness scenarios, and for some specific applications, our metric slightly overestimates the cache space required. This could be addressed by employing an unfairness threshold to enable or disable the policy, but since the increase in unfairness is so small, we decided to keep the policy as simple as possible.

C. *Putting it all together: the Dunn policy*

The previous discussion indicates that a dynamic policy selecting the optimal number of clusters for each workload, and the optimal number of ways for each cluster would be the best approach. This claim led us to design the Dunn partitioning policy, which is the one that provides the best results overall.

Figure 10 shows normalized unfairness over no partitioning for each of the studied 8-application workloads and compares the Dunn policy with the 2K and SF3-4K policies, since both policies achieve a significant reduction in unfairness, with performance results within the same range as Dunn. The workloads are sorted in ascending unfairness order according to the Dunn policy results, and the 2K and SF3-4K results have been plotted following the same order. Clearly, the 2K and Dunn policies are the ones that reduce unfairness the most over no partitioning, by on average 36% and 39%, respectively, and by up to 80%. The other policy, SF3-4K, has less effect on unfairness, reducing it by 16% on average.

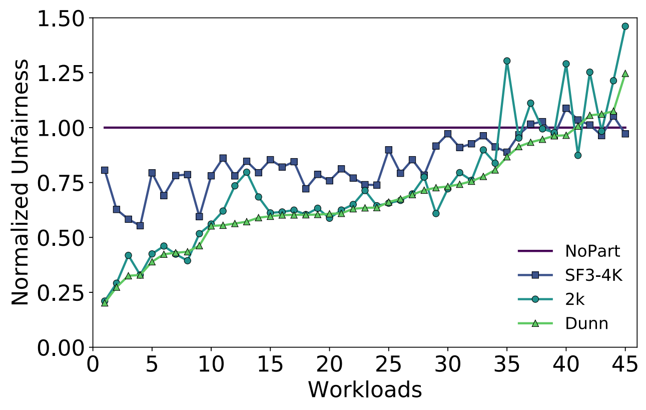

Fig. 10. Normalized unfairness results with respect to NoPart for the 45 8-application workloads with the different partitioning policies.

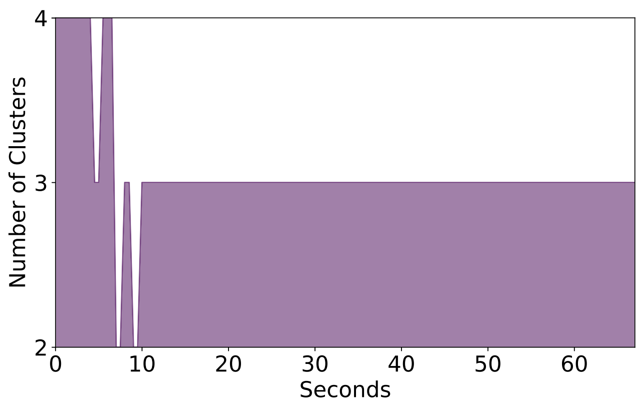

Fig. 12. Number of clusters used during the execution of the 12th 8-application workload.

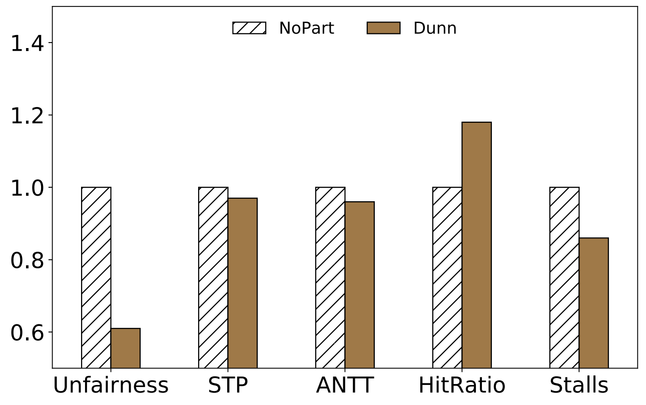

Fig. 11. Average Dunn results normalized against NoPart for the 8-application workloads.

Figure 11 presents the average results for different metrics (see Section II) across all the 8-application workloads achieved by the Dunn policy. The results have been normalized against the NoPart baseline. Note that there is a clear inverse correlation between LLC hit ratio and unfairness. The reason is that most of the time unfairness is caused by the slowest applications, which frequently access the cache but miss due to lack of enough cache space. As a consequence, these applications stall for long periods. As the Dunn policy gives more cache ways to the slowest applications, their accesses start to hit the cache, which reduces the number of stalls and improves system fairness. This behavior also improves per-application performance (ANTT), although the STP metric is slightly reduced because the fastest applications are given fewer resources.

As explained above, the Dunn policy dynamically chooses the optimal number of clusters. To analyze if the improvements that Dunn provides over the mK policies come, in fact, from choosing the *correct* number of clusters, Figure 12 plots the number of clusters used during the execution time of a workload (workload number 12 in Figure 10 and number 11 in Figure 9[3]). According to Figure 9, it seems clear that two clusters does not work well for this workload, and that three and four clusters perform similarly. Figure 10 shows that for this workload, Dunn performs significantly better than 2K, so it must be picking three and four clusters as the optimal number of clusters. As expected, Figure 12 corroborates this.

In summary, a dynamic policy to adapt the number of clusters and the number of ways per cluster can, in fact, provide meaningful fairness improvements compared to static policies. Another insight is that the potential of CAT seems to be limited more by the number of available ways in the LLC than by the number of classes of service.

D. Results with 12 cores

To evaluate the scalability potential of the Dunn policy we now consider 12-application workloads created following the approach described in Section VII. Figures 13 and 14 provide insight regarding how the Dunn policy affects system performance, unfairness and LLC hit ratio.

Looking at Figure 13 it is clear that the important unfairness reduction achieved for the 8-application workloads is also accomplished for the 12-application workloads, where unfairness is reduced by 25% on average. This unfairness reduction does not affect STP, which on average remains the same, but comes accompanied by a significant improvement in ANTT, which decreases by 12% on average. As with 8-application workloads, the unfairness and performance (ANTT) improvements are because the Dunn policy greatly increases the LLC hit ratio, which in turn reduces the number of cycles that cores are stalled (by 15%, on average).

Figure 14 shows detailed results for each one of the workloads. Unfairness is reduced for all the workloads by at least 12%, and up to 45%. ANTT also improves for all the workloads. Finally, depending on the workload, the Dunn policy presents an STP that deviates from NoPart by less than 10%.

[3]While the workloads in both figures are the same, they have been sorted following a different criterion.

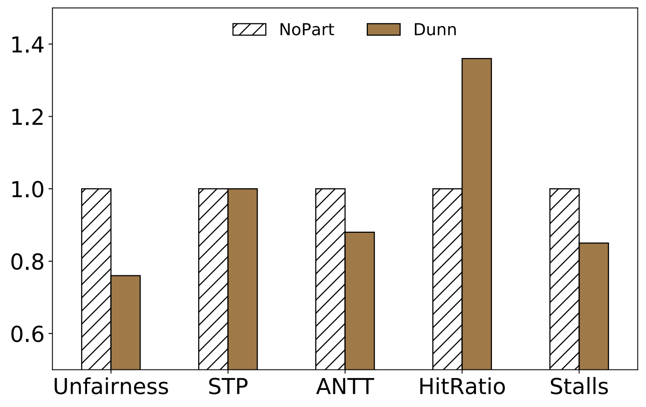

Fig. 13. Average Dunn results normalized against NoPart for the 12-application workloads.

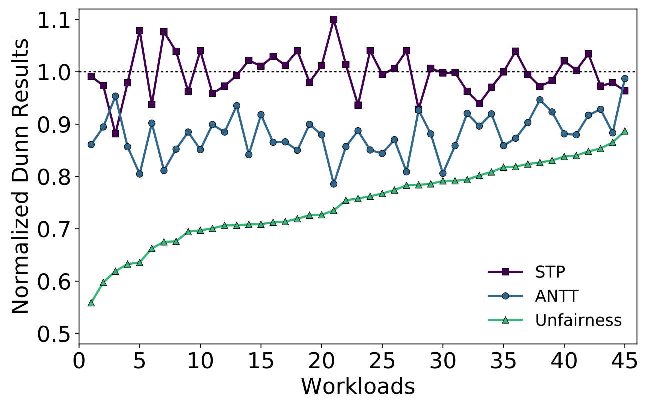

Fig. 14. Dunn results normalized against NoPart for the 45 12-application workloads.

IX. Related Work

A large body of research has concentrated on addressing contention within the LLC, proposing solutions of a very different nature. These proposals can be implemented in hardware or software. In this section we review them, paying special attention to those that make use of CAT. Additionally we also review proposals that estimate cache interference and others whose focus is also on fairness.

a) Online LLC management using CAT: Recent publications make use of Intel's CAT technology to manage the use of the LLC with different aims. Both Heracles [31] and Dirigent [49] focus on maximizing utilization in large-scale datacenters without affecting user-perceived latency in latency-critical applications. To do so, 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 [15] focuses on cloud computing providers that rent virtual machines, and uses a market-driven auction system to partition the LLC into non-overlapping partitions depending on how much each guest is willing to pay and how that affects the rest.

b) Partitioning proposals that require hardware changes: A mechanism for cache partitioning is way partitioning. An example of a prototype multicore using this technique is provided by Cook et al. [5]. There are different ways for managing the number of ways for the different applications. UCP [34] and ASM-Cache [41] use set sampling and duplicate cache tags to gather information that is later used to partition the cache. On the other hand, the proposal by Gupta and Zhou [17] partitions the cache while increasing spatial locality with aggressive prefetching.

Other proposals enforcing cache partitions at the way level change the cache replacement algorithm. This is the case for PriSM [32] which manages cache occupancy of different cores by controlling their eviction probabilities. Similarly, Kahn et al. [28] propose a modified replacement policy to dynamically create two logical partitions, one for clean lines and another for dirty lines, with different eviction probabilities. Futility Scaling (FS) [44] is yet another replacement-based cache partitioning scheme. Its goal is to precisely partition the cache while still maintaining high associativity even with a large number of partitions.

A different approach is used in Vantage [37] and Ubik [27]. Both employ ZCaches [36] to partition the cache. Vantage optimizes partitions to gain performance, while Ubik ensures QoS and improves the performance of batch applications. Iyer [25] presents the CQoS cache management framework, which provides prioritized service to multiple heterogeneous threads sharing a cache structure. Chang and Sohi [3] select multiple partitions and enforce them in a time-sharing manner across multiple epochs within a stable program phase. They propose a QoS metric that modulates the allocated cache space for a given thread.

c) Software-based partitioning proposals: A significant amount of work has been devoted to software-based cache partitioning approaches. Most of it has been based on page-coloring [40], [42], [47], [48], which is used to control where application data is located in the cache. Some of them, such as [30], perform application profiling at runtime and choose the page coloring used. However the overhead of this profiling can be high. Solari et al. [38] use page coloring, but with the novelty of considering an LLC addressing scheme similar to the one used by Intel Sandy Bridge processors. The main disadvantage of page coloring is, however, that when repartitioning occurs, memory pages need to be copied to new locations, so is less versatile than a hardware-based mechanism as the one used in this paper.

d) Interference analysis: Shared cache interference both in CMPs and SMTs is a well-known problem that has been studied by several authors. Eyerman et al. [7] and Du Bois et al. [12] propose an approach to measure this interference by duplicating a fraction of the shared LLC cache tags. Ebrahimi et al. [9] propose to keep track of the lines evicted by the different competing cores using a hash table to estimate interference and use this estimation to enforce fairness. Subramania et al. [41] employ an approach similar to Eyerman et al. to estimate application slowdown due to interference.

e) Fairness: Most techniques aimed at achieving fair overall system performance, which is the goal of this work, are software-based and rely on OS scheduling [14], [45], [46]. Ebrahimi et al. [9] improve system fairness by dynamically adapting the rate at which different cores inject requests into the memory subsystem. Kim et al. [29] improve system fairness by partitioning the shared L2 cache without requiring any OS modification. However, their approach requires offline profiling, which makes it impractical. The approach proposed by Sharifi et al. [39] is based on changing the cache replacement policy to focus on fairness among cores by penalizing the core with the highest IPC in favor of the others.

X. Conclusions

Previous research on cache sharing has shown that partitioning approaches can be used to distribute cache ways among the running applications with the aim of improving fairness. These approaches have been evaluated using simulation methodologies with much simplification over real machines. Moreover, a critical issue to address fairness, which is the need for estimating the slowdown of the running applications, is performed in previous works using extra hardware, which makes them impractical in existing processors.

This work presented a family of cache partitioning policies to address system fairness on commercial Intel processors. All the devised policies employ application clustering to group applications into classes of service using the $STALL\_L2\_PENDING$ event counter, which has been proven in this work to be a good proxy for per-application slowdown. Once the classes of service have been established, each class is given a number of ways according to a simple mathematical function. Experimental results show that for 8-application workloads, the Dunn policy (i.e., the best performing one) reduces system unfairness by up to 80% (39% on average) for 8-application workloads and by up to 45% (25% on average) for 12-application workloads compared to a non-partitioning approach without harming overall system performance (STP) and even significantly improving per-application performance (ANTT) for 12-application workloads.

Finally, we would like to remark that through this work we have presented the proposed policies step-by-step, discussing all the decisions taken on the intermediate phases that led us to the most refined proposal, and showing how to take advantage of the different capabilities of the CAT technology. Regarding CAT from the fairness point of view, two main conclusions can be drawn. First, in most of the evaluated workloads, using only two classes of service during the whole execution time allows achieving outstanding system fairness results. In other words, counterintuitively, using additional classes of service does not always result in further system fairness enhancements. Notice that this observation contrasts with the current Intel trend, which has increased the number of supported classes of service from 4 to 16 in the latest Intel microprocessor generation. Second, instead of using a large number of classes of service, the key issue to deal with system fairness lies on the function employed to assign cache ways to classes of service. We find that an exponential distribution of cache ways is the one that best improves system fairness.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback and suggestions. This work was supported in part by the Spanish Ministerio de Economía y Competitividad and Plan E funds, under grants TIN2014-62246-EXP and TIN2015-66972-C5-1-R. Additional support is provided by the FWO projects G.0434.16N and G.0144.17N.

References

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

[2] 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.

[3] J. Chang and G. S. Sohi, "Cooperative Cache Partitioning for Chip Multiprocessors," in *Proceedings of the 21st Annual International Conference on Supercomputing (ICS)*, 2007, pp. 242–252.

[4] C. K. Chow, "Determination of Cache's Capacity and Its Matching Storage Hierarchy," *IEEE Transactions on Computers*, vol. 25, no. 2, pp. 157–164, 1976.

[5] H. Cook, M. Moretó, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-Efficiency while Preserving Responsiveness," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013, pp. 308–319.

[6] 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 *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 455–456.

[7] K. Du Bois, S. Eyerman, and L. Eeckhout, "Per-thread Cycle Accounting in Multicore Processors," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, pp. 29:1–29:22, 2013.

[8] J. C. Dunn, "A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters," *Journal of Cybernetics*, vol. 3, no. 3, pp. 32–57, 1973.

[9] 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 *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 335–346.

[10] B. Everitt, *The Cambridge dictionary of statistics*, 2002.

[11] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.

[12] S. Eyerman and L. Eeckhout, "Per-thread Cycle Accounting in SMT Processors," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009, pp. 133–144.

[13] S. Eyerman and L. Eeckhout, "Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance," *IEEE Computer Architecture Letters*, vol. 13, no. 2, pp. 93–96, 2014.

[14] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "Addressing Fairness in SMT Multicores with a Progress-Aware Scheduler," in *Proceedings of the 29th International Parallel and Distributed Processing Symposium (IPDPS)*, 2015, pp. 187–196.

[15] L. Funaro, O. A. Ben-Yehuda, and A. Schuster, "Ginseng: Market-Driven LLC Allocation," in *Proceedings of the USENIX Conference on Usenix Annual Technical Conference*, 2016, pp. 295–308.

[16] R. Gabor, S. Weiss, and A. Mendelson, "Fairness Enforcement in Switch on Event Multithreading," *ACM Transactions on Architecture and Code Optimization*, vol. 4, no. 3, 2007.

[17] S. Gupta and H. Zhou, "Spatial Locality-Aware Cache Partitioning for Effective Cache Sharing," in *Proceedings of the 44th International Conference on Parallel Processing (ICPP)*, 2015, pp. 150–159.

[18] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A K-Means Clustering Algorithm," *Journal of the Royal Statistical Society*, vol. 28, no. 1, pp. 100–108, 1979.

[19] A. Hartstein, V. Srinivasan, T. R. Puzak, and P. G. Emma, "On the Nature of Cache Miss Behavior: Is It $\sqrt{2}$?" *Journal of Instruction-Level Parallelism*, vol. 10, 2008.

[20] 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 *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006, pp. 13–22.

[21] Intel Corporation, *Intel Xeon Processor E5-2658 v3*. Available: http://ark.intel.com/es/products/81905/Intel-Xeon-Processor-E5-2658-v3-30M-Cache-2_20-GHz

[22] Intel Corporation, *Processor Counter Monitor*. Available: https://github.com/opcm/pcm.git

[23] Intel Corporation, *User space software for Intel Resource Director Technology*. Available: https://github.com/01org/intel-cmt-cat

[24] Intel Corporation, *Improving Real-Time Performance by Utilizing Cache Allocation Technology*, April 2015, no. 31843-001US. Available: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf

[25] R. Iyer, "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms," in *Proceedings of the 18th Annual International Conference on Supercomputing (ICS)*, 2004, pp. 257–266.

[26] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "QoS Policies and Architecture for Cache/Memory in CMP Platforms," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2007, pp. 25–36.

[27] 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 (ASPLOS)*, 2014, pp. 729–742.

[28] S. M. Khan, A. R. Alameldeen, C. Wilkerson, O. Mutlu, and D. A. Jiménez, "Improving Cache Performance Using Read-Write Partitioning," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 452–463.

[29] S. Kim, D. Chandra, and D. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," in *Proceedings of the 13rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2004, pp. 111–122.

[30] L. Liu, Y. Li, C. Ding, H. Yang, and C. Wu, "Rethinking Memory Management in Modern Operating System: Horizontal, Vertical or Random?" *IEEE Transactions on Computers*, vol. 65, no. 6, pp. 1921–1935, 2016.

[31] 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 (ISCA)*, 2015, pp. 450–462.

[32] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic Shared Cache Management (PriSM)," in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012, pp. 428–439.

[33] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2007, pp. 146–160.

[34] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO)*, 2006, pp. 423–432.

[35] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *Journal of Computational and Applied Mathematics*, vol. 20, pp. 53 – 65, 1987.

[36] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling Ways and Associativity," in *Proceedings of the 43rd Annual International Symposium on Microarchitecture (MICRO)*, 2010, pp. 187–198.

[37] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-grain Cache Partitioning," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 57–68.

[38] A. Scolari, D. B. Bartolini, and M. D. Santambrogio, "A Software Cache Partitioning System for Hash-Based Caches," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 4, pp. 57:1–57:24, 2016.

[39] A. Sharifi, S. Srikantaiah, M. T. Kandemir, and M. J. Irwin, "Courteous Cache Sharing: Being Nice to Others in Capacity Management," in *Proceedings of the 49th Annual Design Automation Conference (DAC)*, 2012, pp. 678–687.

[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 (ICS)*, 1999, pp. 155–164.

[41] 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 *Proceedings of the 48th Annual International Symposium on Microarchitecture (MICRO)*, 2015, pp. 62–75.

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

[43] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware Scheduling on single-ISA Heterogeneous Multi-cores," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 177–188.

[44] R. Wang and L. Chen, "Futility scaling: High-associativity cache partitioning," in *Procedings of the 47th Annual International Symposium on Microarchitecture (MICRO)*, 2014, pp. 356–367.

[45] 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," *Journal on Transactions on Parallel and Distributed Systems*, vol. 26, no. 2, pp. 444–454, 2015.

[46] 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, 2012, pp. 295–306.

[47] Y. Ye, R. West, Z. Cheng, and Y. Li, "COLORIS: A Dynamic Cache Partitioning System Using Page Coloring," in *Proceedings of the 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2014, pp. 381–392.

[48] X. Zhang, S. Dwarkadas, and K. Shen, "Towards Practical Page Coloring-based Multicore Cache Management," in *Proceedings of the 4th European Conference on Computer Systems (EuroSys)*, 2009, pp. 89–102.

[49] H. Zhu and M. Erez, "Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016, pp. 33–47.