Document downloaded from:

http://hdl.handle.net/10251/168885

This paper must be cited as:

Pons-Escat, L.; Sahuquillo Borrás, J.; Selfa, V.; Petit Martí, SV.; Pons Terol, J. (2020). Phase-Aware Cache Partitioning to Target Both Turnaround Time and System Performance. IEEE Transactions on Parallel and Distributed Systems. 31(11):2556-2568. https://doi.org/10.1109/TPDS.2020.2996031

The final publication is available at

https://doi.org/10.1109/TPDS.2020.2996031

Copyright Institute of Electrical and Electronics Engineers

Additional Information

© 2020 IEEE. Personal use of this material is permitted. Permissíon from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertisíng or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Phase-Aware Cache Partitioning to Target both Turnaround Time and System Performance

Lucia Pons, Julio Sahuquillo, Vicent Selfa, Salvador Petit, Julio Pons

Abstract—The Last Level Cache (LLC) plays a key role in the system performance of current multi-cores by reducing the number of long latency main memory accesses. The inter-application interference at this shared resource, however, can lead the system to undesired situations regarding performance and fairness. Recent approaches have successfully addressed fairness and turnaround time (TT) in commercial processors. Nevertheless, these approaches must face sustaining system performance, which is challenging. This work makes two main contributions. LLC behaviors regarding cache performance, data reuse and cache occupancy, that adversely impact on the final performance are identified. Secondly, based on these behaviors, we propose the *Critical-Phase Aware Partitioning Approach (CPA)*, which reduces TT while sustaining (and even improving) IPC by making an effective use of the LLC space. Experimental results show that CPA outperforms CA, Dunn and KPart state-of-the-art approaches, and improves TT (over 40% in some workloads) over Linux default behavior while sustaining or even improving IPC by more than 3% in several mixes.

Index Terms—Cache memories, Multi-core multiprocessors, Memory structures, Memory hierarchy, Performance.

1 INTRODUCTION

APPLICATIONS executing in modern high-performance multi-core processors compete for shared resources. Among these resources, the Last Level Cache (LLC), typically shared among all the cores, plays a key role in the final performance. The common design choice taken by computer architects is to provide a huge LLC, on the order of a few MBs per core, which adds up to tens of MBs. This capacity is much higher when the LLC is built using denser technologies, like in the IBM POWER9 [1] or in the Intel Knights Landing [2] architectures.

Cache sharing allows improving resource utilization and presents important advantages over splitting the area devoted to the LLC into smaller private caches [3]. However, sharing the LLC can lead to important shortcomings from the system performance perspective. The inter-application interference at the LLC makes the system and individual application performance become unpredictable, thus leading to undesired situations for both performance and fairness.

This interference can be addressed in some recent Intel and ARM processors, which provide support to distribute the LLC cache ways among the co-running applications. For instance, Intel's **CAT** technology allows assigning specific cache ways to either groups of applications or cores. Several cache partitioning approaches leveraging Intel CAT have been proposed addressing different performance targets like system fairness [3], turnaround time (TT) [4] and system throughput [5]. These works achieve the objectives they target by, among others, (severely) limiting the LLC space some *non-critical* applications are able to occupy. This way avoids that these applications slow down co-running applications that access the LLC less frequently, but whose performance is more sensitive (critical) to LLC space. Therefore, their performance can significantly impact on the final system performance. Nevertheless, benefiting some specific applications (e.g., to improve fairness) at the expense of damaging the best performing ones negatively impacts on the system throughput.

In a recent work [4], we proposed the *Critical-Aware (CA) Partitioning Approach* aimed at addressing this issue. CA is a conceptually simple but effective approach, that achieves excellent results by dynamically classifying applications in two main groups, namely *critical* and *non-critical* applications. The former category groups applications whose performance is affected (critical) by the amount of cache space, and the latter includes non-affected applications. Then, the LLC is partitioned accordingly by limiting the number of cache ways assigned to non-critical applications and isolating critical applications in a single partition with a larger amount of cache space.

In this work we propose the **Critical-Phase Aware (CPA) Partitioning Approach**, which presents major improvements over the CA approach but follows the same design philosophy. CPA enhances the design of CA in four ways, which are the contributions of this work.

1) An exhaustive characterization study from the LLC perspective is performed, establishing the relationship between an application's LLC behavior and its impact on the system performance. The study analyzes LLC performance in terms of misses per kilo-instruction (MPKI_LLC), data reuse in terms of hits per kilo-instruction (HPKI_LLC), and cache occupancy. To the best of our knowledge, this is the first approach that uses the LLC occupancy to drive the partitioning strategy. Monitoring the occupancy allows our approach to identify applications wasting cache space.
2) We propose the CPA partitioning approach that considers the results of the characterization study to deal with both TT and performance. We found that LLC behavior changes are linked to (IPC) phase

- *Department of Computer Engineering, Universitat Politècnica de València, Valencia, Spain.*
E-mail: lupones@disca.upv.es

changes of the application, therefore CPA is triggered by phase changes instead of at fixed intervals, reducing the overhead over existing approaches.

3) We found that CA does not properly work when the level of *criticality* widely varies among the applications in sharing the critical partition. To deal with this issue, the number of partitions is not limited *a priori*, but additional partitions are used to isolate programs based on their run-time behavior. This provides a finer control of the LLC space assigned to each application.
4) A wider set of applications, a total of 50, taken from both SPEC CPU 2006 [6] and 2017 [7] suites, has been studied in this work; thus broadening the variety of behavioral patterns.

CPA has been evaluated in TT, average normalized turnaround time (ANTT) –which provides some notions of fairness– and system throughput (IPC). Experimental results show that CPA on average outperforms KPart and Dunn state-of-the-art approaches along the three studied metrics.

2 APPLICATION CHARACTERIZATION

This section characterizes in detail the behavior of SPEC CPU 2006 and 2017 benchmark suites, analyzing the relationship between MPKI_LLC (misses per kilo-instruction in the LLC), HPKI_LLC (hits per kilo-instruction), IPC (instructions per cycle) and LLC occupancy. Since there are applications whose name appears in both suites, from now on the suffix `_06` and `_17` will be added to specify the corresponding suite.

A seminal technique that characterizes the LLC behavior of the applications and designs a partitioning approach, assigning a particular management strategy to each category, was presented in [4]. The characterization study identified two main groups of applications: **Critical and Non-critical applications**. In the former group, an increase in the assigned LLC space of these applications results in an IPC increase. Similarly, the lower the number of assigned cache ways the higher the MPKI_LLC these applications experience. In the latter, In contrast, the performance of latter group does not improve by increasing the amount of LLC space beyond two 1 MB ways[1]. Additionally, their MPKI_LLC is particularly low (less than 1), which easily allows identifying these applications at run-time.

Based on the results of the characterization study, the **Critical-Aware Partitioning Approach (CA)** [4] was proposed, a dynamic algorithm that distributes the LLC space according to the applications' needs. This algorithm improves performance for most of the studied workloads mixes. Further insights can be found in Section 3.

2.1 Problematic Applications

A wider and deeper evaluation of the CA approach for an extensive range of workload mixes revealed that some applications do not fit well in any of the two categories (i.e., critical and non-critical). We found that, even though CPA identifies some as critical (high MPKI_LLC), their performance did not improve with larger cache partitions. Moreover, assigning larger partitions to these applications increases the inter-application interference, thus reducing the overall system performance. To deal with this shortcoming, CPA identifies these atypical behaviors at run-time and treats them in a different way. Applications with these behaviors will be referred to as **problematic**.

1. Notice that a 1-way LLC behaves like a direct-mapped cache so performance can drop considerably (due to conflict misses rise).

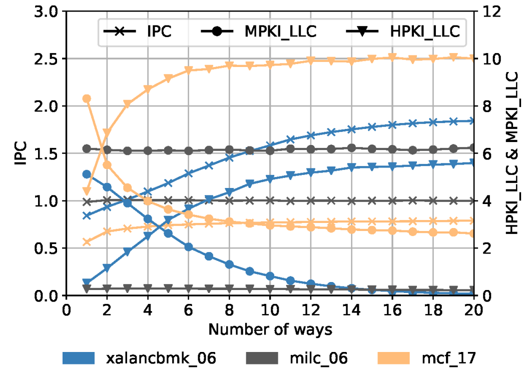


Fig. 1: Average behavior of critical and problematic applications varying the LLC space.

Looking further in this direction, we found that cache block *reuse* can assist to distinguish critical and problematic behaviors. The reuse can be quantified with the hits per kilo-instruction (HPKI_LLC) metric, an additional metric not considered by CA. Figure 1 shows the IPC, MPKI_LLC and HPKI_LLC average values of three applications (`xalancbmk_06`, `milc_06`, and `mcf_17`), all of them identified as critical by CA. For each application, the values shown in the graph were obtained in isolated execution increasing the number of assigned LLC space from 1 to 20 ways (the entire cache). As it can be observed, all of them present an average MPKI_LLC higher than 2 (an average value expected in critical applications). With the only exception of `xalancbmk_06`, the IPC does not significantly increases beyond 2 ways. For instance, `milc_06`'s IPC slightly improves initially with 2 ways and then remains constant. Similarly, the IPC improvement of `mcf_17` is scarce compared to that obtained by `xalancbmk_06` with larger partitions. Therefore, `milc_06` and `mcf_17` can be classified as problematic applications.

Looking at the HPKI_LLC in Figure 1, we can make three interesting observations. Firstly, `xalancbmk_06`'s HPKI_LLC grows with the number of LLC ways, and, conversely, the MPKI_LLC decreases. As in other critical applications, this means that the performance (IPC) achieved by `xalancbmk_06` improves with the amount of LLC space. Secondly, in contrast, `milc_06`'s HPKI_LLC is always almost constant and close to 0, regardless of the number of LLC ways that it has been assigned. This is because LLC blocks are scarcely reused (accessed again) or not reused at all before being evicted. Consequently, the performance improvements are negligible with additional cache space. In this work, refer to this kind of problematic behavior as **squanderer**. Other contemporary works [5], [8], [9], [10] also identify this type of cache polluter behavior, however, in this work applications with this behavior are treated differently.

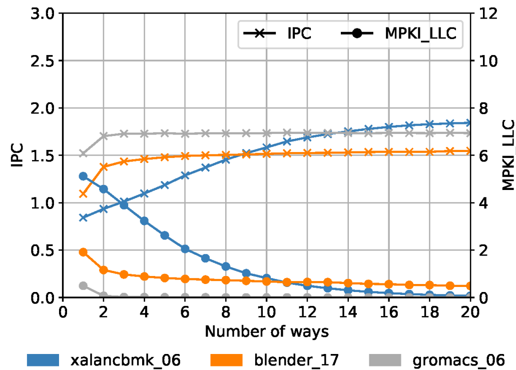


Fig. 2: Average behavior of critical and non-critical applications varying the LLC space.

Finally, mcf_17 presents the highest HPKI_LLC, which increases when the application is granted additional LLC space. Nevertheless, assigning more cache ways over a given number does not improve its performance. The reason is that the out-of-order execution is not able to hide the latency of most of accesses to the LLC, which eventually causes stalls and prevents further performance gains. Moreover, there is an important number of LLC misses, even with large LLC partitions, probably due to mcf_17's cache access patterns being difficult to predict and prefetch effectively. This kind of problematic behavior will be referred to as **bully** and, to the best of our knowledge, it has not been identified in any previous work.

2.2 Degrees of Criticality

Taking a closer look to the behavior of critical applications as initially identified by the CA algorithm, we found that some of them presented a more relaxed behavior; that is, they showed a higher IPC and lower MPKI_LLC than other more critical applications. That is, different degrees of *criticality* can be appreciated. Figure 2 illustrates this fact by plotting the average IPC and MPKI_LLC of two critical applications (xalancbmk_06) and (blender_17), and, for comparison purposes, a non-critical application (gromacs_06). As in Figure 1, the values were obtained in isolated execution increasing the number of assigned LLC ways.

In contrast to gromacs_06, whose performance is not affected, both critical applications present a significant performance degradation with just 2 cache ways (2 MB). Nevertheless, compared to xalancbmk_06, which experiences a really poor IPC (less than 1) with this small cache space, blender_17 presents a mild IPC (around 1.4). In addition, blender_17's IPC stabilizes much earlier (with 6 LLC ways) compared to xalancbmk_06. Therefore, we claim that blender_17 is less critical because: i) it presents higher IPC than other critical applications with reduced LLC space, and ii) it does not require as much LLC space to maximize its performance. This means that it can be assigned to a smaller LLC partition with a minor impact on its performance. This way, more LLC space could be assigned to improve the performance of other applications. Consequently, CPA must ensure that less critical applications do not occupy more LLC space than needed for performance by monitoring their *LLC occupancy* (see further details in Section 2.6). This less critical behavior will be referred to as **medium**, while the more critical behavior has been named **sensitive**.

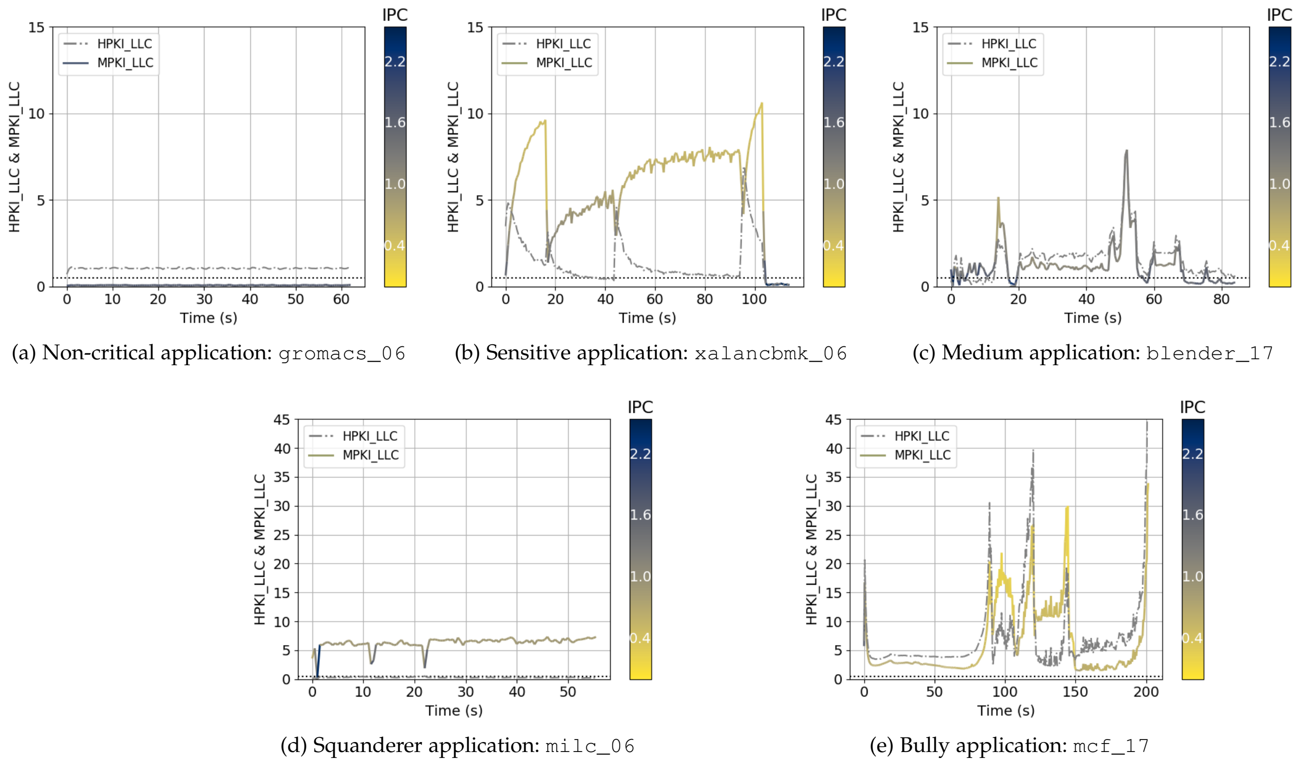
2.3 Dynamic Behavior of Applications

This section shows how applications may present different phases, by executing them in isolation with 2 cache-ways, the minimum LLC space assigned to an application in this work (1 cache-way is not considered due to it behaves as a direct-mapped cache which results in poor performance). This analysis assist us to observe how applications behave in each execution phase with limited (and available for itself on average) LLC space.

Figure 3 illustrates the dynamic behavior of 5 applications, each representative of a category. The MPKI_LLC metric is depicted with a color-changing line and the HPKI_LLC with a dashed gray line. The column at the right side labelled as IPC shows the colormap associated with the IPC values. We found that a MPKI_LLC and HPKI_LLC value below 0.5 means that the LLC does not negatively impact the IPC and that there is a negligible data reuse, respectively. Thus, a horizontal dotted black line at Y=0.5 is plotted in each graph to facilitate the analysis. The three upper plots of Figure 3 correspond to gromacs_06, xalancbmk_06 and blender_17, which show representative non-critical, sensitive and medium behaviors, respectively. The left-most graph corresponds to a typical non-critical behavior. Gromacs_06 presents a MPKI_LLC close to 0 throughout the whole execution, which yields an IPC relatively high (around 2), despite the limited cache space. The next two graphs illustrate sensitive and medium behaviors, respectively. As observed, xalancbmk_06's IPC decreases as the MPKI_LLC increases, that is, a rise in the MPKI_LLC line matches a color change to a lighter color. Across all its execution, this sensitive application presents a relatively low IPC (below 1) and a MPKI_LLC as high as 11.

In contrast, blender_17, the medium application, presents a higher IPC than xalancbmk_06. Following the same trend, blender_17 has lower MPKI_LLC than xalancbmk_06, although it can be considered high if compared with the MPKI_LLC of non-critical applications which is close to 0. Thus, we can argue that with a reduced cache space, the impact on performance is much lower for a medium application than for a sensitive one. On the other hand, the HPKI_LLC metric is used to distinguish a critical (sensitive or medium) from a squanderer behavior. This is done by checking that there is at least some LLC reuse, that is, the HPKI_LLC is higher than 0.5 (dotted line at y=0.5).

The two lower plots of Figure 3 depict the two different behaviors shown by problematic applications. The left-side graph (Figure 3d) presents the behavior of milc_06, a squanderer application with low LLC reuse. As observed, it has a high MPKI_LLC (similar to the sensitive application xalancbmk_06) but its HPKI_LLC is always lower than 0.5. This behavior is homogeneous throughout the execution. The right-side graph (Figure 3e) illustrates the behavior of mcf_17, an application that presents a bully behavior. Notice that this application does not present a bully behavior during the whole execution but only in several execution *phases*. For instance, it can be observed that from approximately second 85 to second 150, both the MPKI_LLC



(a) Non-critical application: `gromacs_06` (b) Sensitive application: `xalancbmk_06` (c) Medium application: `blender_17`

(d) Squanderer application: `milc_06` (e) Bully application: `mcf_17`

Fig. 3: Dynamic behavior of the different application behaviors with 2 cache ways.

Application Category	IPC	LLC MPKI	LLC HPKI
Sensitive	L (<1.3)	H ($Eq.2$ [2])	not VL (≥ 0.5)
Medium	M (≥ 1.3)	H ($Eq.2$ [2])	not VL (≥ 0.5)
Bully	VL (≤ 0.6)	VH (≥ 10)	VH (≥ 10)
Squanderer	$\times$	H ($Eq.2$ [2])	VL (<0.5)
Non-critical		*otherwise*	

TABLE 1: Thresholds for each metric and level ($th_{Metric,Level}$) used to identify the categories. Columns define the metric and H (high), M (medium), L (low), VH (very high) and VL (very low) define the levels.

and HPKI_LLC are dramatically high (both are above 10 most of the time), and thus, the IPC value drops down to about 0.5. In this type of phase, the performance of `mcf_17` would not significantly improve if more cache ways were assigned to it. This is due to the high amount of time taken by the LLC accesses, as explained in Section 2.1. Therefore, even though the amount of LLC hits is very high, this bully application will inevitably achieve poor performance.

2.4 Estimating Thresholds in Multi-Program Execution

While a 2 cache-way configuration –as the assumed in the previous section for studying the dynamic behavior in isolation– provides a relatively low amount of cache space, it could be even lower in multi-program execution, as in this case, the cache space is shared with other co-runners. This means that, although the characterization analysis studies the intrinsic behavior of each application, the threshold values should be determined empirically in multi-program execution.

In this work, thresholds (upper and/or lower) were empirically determined through thousands of experiments for three main metrics: IPC, MPKI_LLC and HPKI_LLC. For each metric, different levels have been defined, referred to as Very High (VH), High (H), Medium (M), Low (L), and Very Low (VL). Table 1 summarizes the values of the thresholds used to perform the experiments presented in Section 6. From now on we will use the term $th_{Metric,Level}$ to refer to the threshold of a given level for given a metric. Notice that threshold $th_{IPC,L}$ behaves as an upper threshold for medium applications and as a lower threshold for sensitive applications. All metrics have a fixed numeric threshold except for MPKI_LLC, which is determined by Equation 2 (Section 4.2), a variation of Equation 1 (Section 3.1) which was used in CA.

2.5 Detecting Phase Changes

We found that *IPC phases* are linked to different LLC space needs or *LLC phases*. For instance, notice that `mcf_17's` behavior, studied above as example of a bully application, does not show this behavior during the whole execution. In the execution phase that extends from the second 5 to the second 75 approximately, `mcf_17` behaves as sensitive. In this phase, `mcf_17` has lower HPKI_LLC and MPKI_LLC values than in the plotted bully phase. Similarly, a non-critical behavior appears at the end of the execution of both `xalancbmk_06` and `blender_17`. Table 2 classifies the studied applications according to their dominant behavior.

Techniques dealing with phase-change detection have been widely studied in the past [11] [12] [13]. The main

2. See Section 4.2 for further details.

Critical Applications	
Sensitive	**Medium**
gcc_17, omnetpp_06, omnetpp_17, soplex_06, xalancbmk_06, xalancbmk_17	blender_17, cactuBSSN_17, fotonik3d_17, GemsFDTD_06, parest_17, roms_17, sphinx3_06, zeusmp_06
Problematic Applications	
Squanderer	**Bully**
milc_06	mcf_06, mcf_17
Non-critical Applications	
astar_06, bwaves_06, bwaves_17, bzip2_06, cactusADM_06, calculix_06, cam4_17, dealII_06, deepsjeng_17, exchange2_17, gamess_06, gobmk_06, gromacs_06, h264ref_06, hmmer_06, imagick_17, leslie3d_06, lbm_06, lbm_17, leela_17, libquantum_06, nab_17, namd_06, namd_17, povray_06, povray_17, perlbench_06, perlbench_17, sjeng_06, tonto_06, wrf_06, wrf_17, x264_17	

TABLE 2: Categorization of SPEC CPU 2006 (_06) and SPEC CPU 2017 (_17) applications.

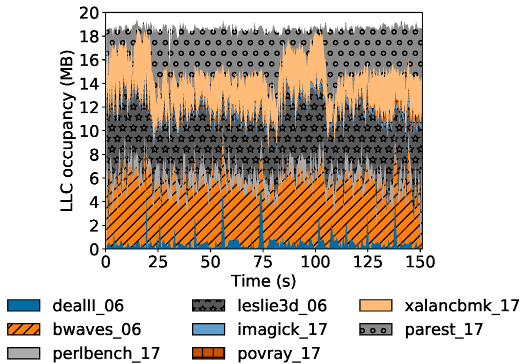


Fig. 4: Dynamic LLC occupancy of mix #17 with NP.

goal of detecting a new phase is to select the architectural configuration leading to optimal performance for the incoming phase. In particular, in this work we use phase-change detection to select the best cache partitioning scheme. We have devised an approach to detect phase changes based on the method proposed by Liao et al. [14], *Interval Coefficient of Variation (ICOV)*, as part of an online phase detection scheme that guides dynamic L2 cache partitioning, implemented using page coloring. ICOV was chosen for two main reasons, simplicity and effectiveness, as well as due to hardware constraints since only four hardware performance counters are available in our processor. This method measures the homogeneity of a given sample of numbers (IPC values in this work). The lower the ICOV value, the closer the IPC of the current interval is to the phase trend. We found 20% as the best performing threshold for our proposal and platform. See [14] for further details.

2.6 Dealing with LLC Occupancy

From the analysis of the defined categories we can summarize the occupancy requirements of each category. Non-critical applications require almost no occupancy and present low interference, so they can be placed together in a single partition. Critical applications have significant LLC space requirements, but it is important to distinguish sensitive from medium, since the latter does not need so much space. Squanderer applications have little reuse, and therefore, they hardly require any space like non-critical applications. Finally, bully applications need a minimum amount of LLC space in order to not degrade even more their performance.

Figure 4 shows an example of the LLC space occupied by each application of one of the studied mixes (#17) under the default Linux scheduler, i.e., when no partitioning (NP) policy is applied. This mix contains 1 medium application (`parest_17`), 1 sensitive application (`xalancbmk_17`), and 6 non-critical applications. Two observations can be made. Firstly, there are non-critical applications like `bwaves_06` and `leslie3d_06` that occupy more space than the critical ones. Secondly, `parest_17` occupies the same (or even more) space than `xalancbmk_17`, in spite of `xalancbmk_17` having higher space requirements. Notice that CPA correctly address both observations by properly identifying the different cache behaviors.

From this example we can conclude that the LLC occupancy metric cannot be used alone to classify applications into the categories previously defined, but other LLC-related metrics like IPC, MPKI and HPKI, should be used instead. Nevertheless, we can still leverage the LLC occupancy of individual applications to check if, once classified, they are making an efficient use of the cache space. For instance, in the previous example, the LLC occupancy of a non-critical application like `bwaves_06` could be monitored to check if it is using an excessive amount of cache space. Likewise, by monitoring LLC occupancy and performance (IPC) of critical applications, we can detect a wasteful usage of LLC space performed by some medium applications, which do not need as much LLC space as sensitive applications. Further details can be found in Section 4.3.

3 Critical-Aware Approach

To make this paper self-contained, this section summarizes the Critical-Aware (CA) Partitioning Approach [4]. The main aim is to improve system throughput by using only two cache partitions or classes of service (CLOS) (see Section 5): one for critical applications and another for non-critical ones. CA consists of three main phases: i) application classification, ii) base partition settings and, iii) dynamic adjusting of partitions, discussed next.

3.1 Application Classification

At the beginning of the execution, before carrying out any action, some time is taken to warm-up the cache. After that, the algorithm enters the *reset state*, in which the MPKI_LLC of all the applications is computed. The algorithm computes the rolling mean (μ) and standard deviation (σ) for the last 10 time intervals of the MPKI_LLC, considering all the applications to detect outliers (that is, critical behavior) by using Miller's criterion [15]. Applications outliers are detected comparing the MPKI_LLC of the current interval to the result of Equation 1.

$$Limit_outlier_MPKI_LLC = \mu + 3 \times \sigma. \quad (1)$$

CA cache configurations (# of used ways)		
# Critical Apps.	**CLOS #1 ways (mask)**	**CLOS #2 ways (mask)**
1	10 (`0x003ff`)	12 (`0xfff00`)
2	9 (`0x001ff`)	13 (`0xfff80`)
3	8 (`0x000ff`)	14 (`0xfffc0`)
0 or more than 3	20 (`0xfffff`)	20 (`0xfffff`)
CPA extensions		
App. Type	**# CLOS(es)**	**# of Ways (mask)**
Bully/Non-critical	1	Same as CA
Critical	2, 3 or 4	Same as CA
Squanderer	5 or 6	2 ways/CLOS shared with CLOS # 1 (`0x00003` with 1 CLOS, `0x0000f` with 2 CLOS)

TABLE 3: Initial cache mask configurations for CA and extended configurations used in CPA.

3.2 Base Partition Settings

Once applications are classified, the algorithm creates two partitions (CLOS #1 for non-critical and #2 for critical applications), whose sizes depend on the number of critical applications detected. The higher the number of critical applications, the more cache ways are assigned to CLOS #2, except when no critical applications are found. Similarly, when there is a majority of critical applications, the cache is configured as a single partition (default configuration).

The initial partitions' layouts, listed on the top part Table 3, were empirically determined based on a deep and thorough study of static configurations, evaluating application mixes with different numbers of critical applications. Notice that a partition size represents a given percentage of the total LLC ways. For instance, with 1 critical application, 60% of the ways are allocated to the critical application's CLOS and 50% to the non-critical applications' CLOS, having 10% of the cache ways shared between both CLOS. Thus, applying this observation the proposed approach could be easily generalized and adapted to another CAT machine deploying an LLC with different number of ways. The size of the partitions is then dynamically adjusted at run-time as discussed in Section 3.3. When the number of critical applications varies due to a change in the behavior of an application (from critical to non-critical or vice-versa), the actual partitioning scheme must be updated. To this end, the algorithm transitions back to the reset state setting the default cache configuration (all applications in CLOS #1 with 20 ways). Then, the classification process starts again.

3.3 Dynamic Adjusting of Partitions

The layouts (i.e., bit masks) of the initial partitions are dynamically adjusted by CA at run-time by following a finite state machine (see [4] for further details). Upon a state change, CA measures the impact of this change on the system performance and checks if it matches the expected one to provide feedback to the partitioning policy.

An issue CA faces when dynamically adjusting partitions is that some applications tend to have phases in which they are critical and phases where they are not. This is challenging because CA resets the partitioning when a change in the number of critical applications occurs. A high number of resets can reduce the potential system throughput. CPA is a phase-change driven approach, so it deals with this drawback.

Algorithm 1 CPA pseudo-code

```
1:  ———————— STEP 1 ————————
2:  for all apps do
3:     Read I, C, and LLC events (#misses, #hits, occupancy)
4:     Compute metrics: IPC, MPKI, HPKI
5:  end for
6:  ———————— STEP 2 ————————
7:  if First Interval then
8:     update_clos = true
9:  else
10:    for all apps do
11:       Compute ICOV
12:       if ICOV > ICOV_threshold & app. behavior change then
13:          update_clos = true
14:       end if
15:    end for
16: end if
17: if update_clos then
18:    Update cache settings according to Table 3
19:    return
20: end if
21: ———————— STEP 3 ————————
22: for all critical apps do
23:    if IPC > th_IPC,L & LLCoccup > LLCoccup_critical/2 then
24:       Reduce the #ways assigned to the CLOS
25:    end if
26: end for
27: if NumCritical == 1 & NumMedium == 1 then
28:    Enlarge CLOS #1 to leverage the free space
29: end if
30: ———————— STEP 4 ————————
31: for all non-critical apps do
32:    if LLCoccup > LLC_CLOS1/3 &
       MPKI < 0.5 & HPKI < 0.5 then
33:       Isolate app. in a CLOS with few ways
34:    end if
35: end for
36: ———————— STEP 5 ————————
37: if NumCritical ≥ 1 then
38:    Adjust cache sizes of CLOS #1 and critical CLOS(es) like CA
39: end if
```

4 Critical Phase-Aware Proposal

4.1 General Overview of the Approach

CPA, like CA, devotes the first intervals of execution to warm up the cache. Initially, all applications are assumed to be non-critical and allocated in CLOS #1, which spans the whole cache (i.e., the default CAT configuration).

Algorithm 1 depicts the pseudo-code of CPA partitioning policy which is applied (after the warm-up phase) periodically in each time interval of the execution. Firstly, in **step 1**, the hardware performance counters are read and the collected data are used to calculate the inputs to the algorithm. Five main hardware events are sampled for each studied application: instructions (I), cycles (C), and three LLC events ($\#misses, \#hits$, and $occupancy$). The gathered values are used to compute the inputs of the algorithm (MPKI_LLC, HPKI_LLC, IPC, and LLC occupancy).

In **step 2**, the ICOV value of each application is computed in order to detect phase changes, which are detected when the ICOV surpasses the $ICOV_{threshold}$ value. In case a phase change is detected, it is checked if a change in the application behavior has also occurred. This is done comparing

the inputs of the algorithm with the thresholds presented in Table 1. For instance, one application is categorized as bully if its IPC is very low (less than $th_{IPC,VL}$), and both its MPKI_LLC and HPKI_LLC are very high (greater than $th_{MPKI,VH}$ and $th_{HPKI,VH}$, respectively). If a change in the application behavior is found, the cache configuration is updated according to the bottom part of Table 3. This configuration update may imply moving the application from one CLOS to another (e.g., from CLOS #1 to CLOS #2) and/or updating the bit masks or number of cache ways assigned to one or more CLOS.

The next steps further refine the cache configuration by determining if the applications assigned to the cache partitions are *behaving properly* regarding their LLC occupancy. That is, CPA checks if an application is using more cache space than it needs. In **step 3**, critical applications are checked for a medium behavior. Since these applications do not need so much LLC space as sensitive applications (see Section 2.2), CPA checks if there is a critical application that shows a medium behavior (IPC is higher than $th_{IPC,L}$) and occupies too much cache space (more than half of the cache space occupied by the critical applications $LLCoccup_{critical}$). In such a case, the number of ways assigned to the CLOS holding the medium application are reduced to the proportional part. That is, half if there is one or two critical applications and one third in case there are three critical applications. Given that each critical application resides individually in one CLOS (see Section 4.3 for further details), the space assigned to each critical application can be easily managed. In case there is only one critical application and it is detected as medium, the number of ways of CLOS #1 are increased so no cache ways are left unused. Note that, in case there are more than one critical application, at least one should be considered as sensitive since halving the space to, for instance, two critical applications will leave too little space for them, damaging their performance.

In **step 4**, non-critical applications are checked. In this case, CPA isolates applications that occupy an excessive amount of LLC space (quantified as more than one third of CLOS #1's space) and that make no profit of it. That is, they show a very low reuse (HPKI_LLC) and misses per kilo-instruction (MPKI_LLC). These applications are isolated in a separate CLOS with few cache ways shared with CLOS #1. This cache arrangement avoids that these cache-greedy applications occupy too much cache space.

Finally, in **step 5**, the partition sizes are adjusted as done in CA. This mechanism has been implemented in a way that it does not let critical applications take too much space and confine the remaining applications to a marginal space. Remark that every time the cache configuration is updated in this step, CPA waits some idle intervals where it is not adjusted again, leaving some time for applications to take advantage of the additional space or reduce the amount they are using to match the new configuration.

4.2 Identifying LLC Behaviors at Run-Time

LLC behaviors, summarized in Table 1, are checked as follows. First, the algorithm checks for a bully behavior. As discussed above, applications presenting this behavior exhibit very high MPKI_LLC and HPKI_LLC values (higher than $th_{MPKI,VH}$ and $th_{HPKI,VH}$, respectively), and very low IPC (less than $th_{IPC,VL}$). Since the performance of these applications does not improve by assigning them a higher amount of cache ways, two solutions could, in principle, apply, i) isolate it in a single and small CLOS or, ii) allow it to remain in CLOS #1 together with non-critical applications. We evaluated both design choices and found that the second choice provides the best results, probably because they span to a higher number of cache ways.

Second, CPA checks for critical (sensitive and medium) behaviors. Notice that both sensitive and medium behaviors are identified as critical in step 2, but they are differently addressed in step 3, where the LLC space is adjusted accordingly. Critical applications present a high MPKI_LLC (greater than $th_{MPKI,H}$). Note that in Table 1 threshold $th_{MPKI,H}$ is not defined by a fixed value but by Equation 2. Therefore, this threshold varies depending on the benchmarks that make up the mix but, according to the equation, the threshold value will be always in the *high* level range (i.e., > 1). Although many statistical studies use $3 \times$ std over the μ, we found empirically that a more relaxed threshold (1.5 standard deviations) works well across the studied mixes. Additionally, the equation excludes the MPKI_LLC of applications showing a previously detected problematic behavior from the calculation of the mean and standard deviation, since such high values skew the data model. Note too that CPA does not only consider the MPKI_LLC to detect a critical behavior but also takes into account the achieved IPC and the LLC reuse (i.e., HPKI_LLC) (see Section 2).

$$th_{MPKI_LLC,H} = max(1, \mu + 1.5 \times std) \qquad (2)$$

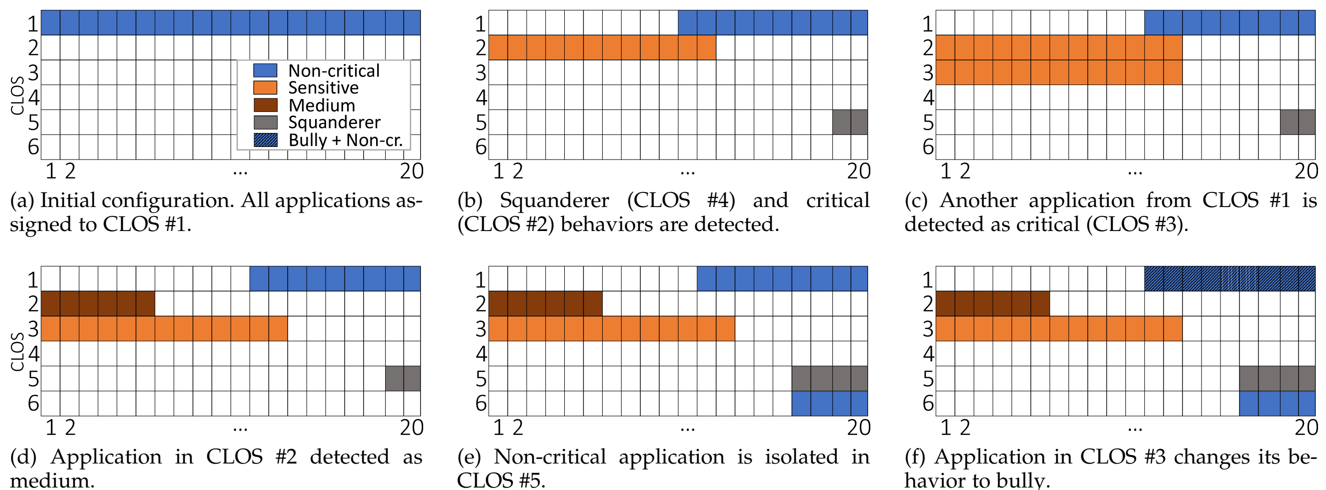
Third, CPA checks for squanderer behavior. An application exhibits this behavior whenever it fulfills two conditions: i) it occupies a significant fraction of the LLC and, ii) it presents a low reuse. The former means that the application experiences a high MPKI_LLC (fulfills Equation 2) , and the latter that it has a very low HPKI_LLC (less than $th_{HPKI,VL}$). Taking into account the previous rationale, CPA isolates squanderer applications into a separate CLOS with few cache ways (shared with CLOS #1) since no performance benefits are achieved with additional space.

Finally, if the behavior of a given application does not fulfill the criteria of any of the three mentioned behaviors, then it is considered non-critical.

4.3 Dynamic CLOS Management: Allocation, Releasement and Adjustment

CPA leverages the data collected from the *LLC occupancy* hardware counter, which measures the cache space occupied by each application. To the best of our knowledge, this is the first time this metric is used to drive a partitioning strategy.

When applications are placed together in a CLOS, the space available is often not shared evenly. Applications present different access rates and the LLC replacement algorithm is driven, among others, by the access rate. This means that if two applications sharing the same CLOS have widely different access rates, the application accessing with higher frequency will likely occupy much more space than the other. Having an unconstrained number of CLOS allows CPA to use private CLOS to host individual applications



(a) Initial configuration. All applications assigned to CLOS #1.

(b) Squanderer (CLOS #4) and critical (CLOS #2) behaviors are detected.

(c) Another application from CLOS #1 is detected as critical (CLOS #3).

(d) Application in CLOS #2 detected as medium.

(e) Non-critical application is isolated in CLOS #5.

(f) Application in CLOS #3 changes its behavior to bully.

Fig. 5: Cache partitioning example in CPA. Each column represents one cache way and each row one CLOS.

and easily control the cache ways assigned to them. Note that a private CLOS does not necessarily means private space, since the ways assigned to a CLOS may be shared with another CLOS.

Private CLOS in CPA serve three main purposes, i) limit the interference between medium and sensitive applications, ii) avoid unfair space distribution among non-critical applications and iii) isolate squanderer applications. The first purpose is achieved by placing each critical application in a specific CLOS. The rationale behind this design choice is to facilitate reducing the LLC space to medium applications since they need less space than sensitive applications, so reducing the inter-critical application interference. The second aim refers to non-critical applications. Even though these applications have little space requirements, some non-critical applications occupy more cache space than they need, i.e., the same performance is achieved with less occupancy. This may affect the performance of co-runners in case they they are left too little space (e.g., less than 1 MB). Thus, if this situation is detected, these applications are isolated in a separate CLOS with a few cache ways. Finally, the third aim is achieved by isolating squanderer applications individually in private CLOS with few LLC ways, since these applications have little reuse and barely need LLC space. Notice that the space assigned to these CLOS is shared with CLOS #1, unlike previous works which isolated this cache pollutant applications in a private CLOS with private ways (not share ways with other co-runners). This fact, however, reduces the effective cache space that can be accessed by the remaining applications, which is is not the best design choice for performance [3]. Therefore, unlike previous works, we allow the ways assigned to squanderer applications to overlap with other applications. In particular, with the ways assigned to non-critical applications and other problematic applications, which are less affected by LLC interference than critical applications.

4.4 Working Example

To help understand how CPA works and illustrate how cache partitions are disposed, this section presents a working example on a hypothetical execution scenario considering a mix made up of eight applications. The example studies six different events that occur along the execution. Figure 5 shows the active CLOS (following the criteria shown in Table 3) and the LLC ways (from 1 to 20 ways) associated to each CLOS on each event.

At the start of the execution (Figure 5a) all the 8 applications of the workload mix are in CLOS #1, which spans all the cache (default cache configuration). After warming up the cache (at *First Interval*), the behavior of each application is checked. Let's suppose that one of the 8 applications exhibits a critical behavior and another one a squanderer. Both of these applications are assigned each to a separate CLOS, i.e., two new cache partitions sized with the initial settings are created (see Figure 5b). In this case, CLOS #2 holds the critical application and is assigned 12 cache ways (ways #1 to #12, see Table 3), and the squanderer application is allocated in CLOS #5 with 2 cache ways (ways #19 and #20). The partition sizes are then dynamically adjusted depending on their LLC requirements. For simplicity, the dynamic adjustment is not shown in this example, since it follows a complex state machine (see [4] for further details). Some cache ways are shared (i.e., overlapped) among CLOS #1 and #2 in order to improve the cache efficiency.

In the next event, Figure 5c, a non-critical application in CLOS #1 experiences a phase change and starts showing a critical behavior. Then, CPA creates a new partition (associated to CLOS #3) to host this application and the CLOS mask is updated. Notice that each application showing a critical behavior is placed on a different private CLOS, but initially both partitions share the same ways. Lets assume now that the critical application in CLOS #2 shows a higher IPC than $th_{IPC,L}$ and it is occupying a high fraction of the critical LLC space. This application is then labeled as medium, and the CLOS #2 size is reduced to half (Figure 5d). The next event (Figure 5e) assumes that an application in CLOS #1 starts showing a squanderer behavior (wasting the cache space by occupying a high fraction of the partition) but it presents a very low MPKI and reuse. To counteract this situation, CPA creates a new partition (associated to CLOS

#6) to isolate this application. Notice that CLOS #5 and #6 are assigned 4 shared ways instead of 2 cache ways each, in order to improve cache efficiency.

Finally, the critical (i.e., sensitive) behavior of the application in CLOS #3 moves to a bully behavior (Figure 5f). When this happens, two main actions are carried out. Firstly, the application in CLOS #3 is returned back to CLOS #1. Secondly, CLOS #2 is given back the space it had before halving it and, in forthcoming intervals, this application will be checked for a medium behavior again.

5 Experimental Framework

The experiments have been conducted in an Intel Xeon E5-2620 v4 processor, with 8 SMT cores running at 2.20 GHz. It has a 20-way 20MB (1MB/way) LLC that supports CAT with 16 CLOS. To carry out the experiments we have developed a framework that i) measures performance using a library based on *Linux perf* [16], and ii) partitions the cache using the primitives provided by the Linux 4.11 kernel.

Intel CAT allows assigning a given amount of LLC ways to either a set of cores or applications, that is, it allocates processor identifiers (PIDs) or logical cores to CLOS. For each CLOS, the user specifies the applications or logical cores assigned to the CLOS, and which cache ways are written to the CLOS, defined with a *capacity bitmask* (CBM).

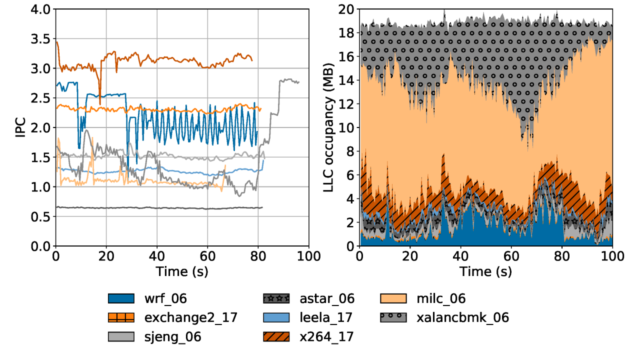
5.1 Workload Mixes

The workload mixes evaluated in this work were randomly generated using 50 applications: 28 applications from the SPEC CPU 2006 benchmark suite and 22 applications from the SPEC CPU 2017 suite. Table 2 shows how applications classify in the defined categories. It can be appreciated that non-critical applications dominate both benchmark suites. Taking this observation into account, 31 mixes consisting of 8 applications each (i.e., the number of cores in the system) were randomly generated keeping non-critical applications as the dominant group, and varying the number of critical and problematic applications from 1 to 3.

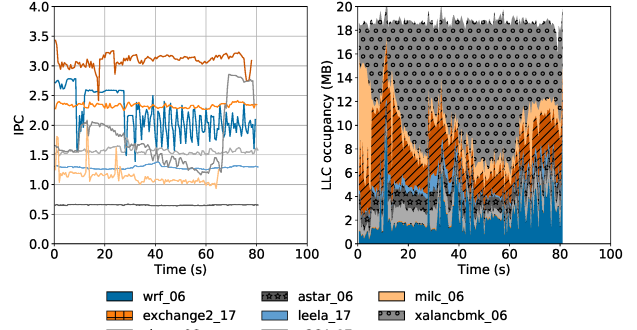
Mixes have been ordered according to the number of applications that the mix contains belonging to the critical or problematic categories; that is, the higher the mix number the higher the number of applications from these categories. Mixes #1 to #12 contain one critical or problematic application, mixes #13 to #24 contain two, and mixes #25 to #31 contain three.

5.2 Experimental Parameters

In addition to the thresholds in Table 1, we set the $ICOV_{threshold}$ to 0.20, the number of *idle* intervals to 5 and the number of *warm-up* intervals to 10. Experiments use CLOS #1 to host non-critical applications, and a new CLOS is allocated whenever a new behavior that requires a private partition is detected. For the 8-application mixes used in this work, CPA considers a maximum of 6 CLOS: *CLOS #1* devoted to non-critical and bully applications, *CLOS #2, #3 and #4* for critical applications (CPA supports up to 3 critical applications), and *CLOS #5 and #6:* for squanderer and non-critical applications wasting LLC space, respectively. Experiments were also performed using more CLOS but results did not improve, thus the presented results only used 6 partitions.



(a) CA



(b) CPA

Fig. 6: Dynamic values of IPC and LLC occupancy of mix #20 under CA and CPA.

5.3 Methodology

To calculate the IPC, MPKI_LLC, HPKI_LLC and LLC occupancy values, we use the following Perf hardware counters: `instructions`, `ref-cycles`, `mem_load_uops_retired.l3_hit`, `mem_load_uops_retired.l3_miss` and `intel_cqm/llc_occupancy/`.

Workload mixes are run until all the applications in the mix have completed a fixed number of instructions. This number corresponds to the number of instructions the application executes when running alone for 60 seconds. When an application reaches this limit, and it is not the last one in the mix to reach such limit, it is restarted so the results of the other applications are not skewed by the fact that they have fewer co-runners. Nonetheless, when analyzing the results, only the values of the first execution run of each application are considered. At regular 500ms intervals, the experimental framework reads the performance counters and passes the values to the partitioning algorithm. Each experiment is repeated 3 times, and the average values and the standard deviation for each metric are derived. Results in Section 6 are shown within a 95% confidence interval with a margin of error lower than 3%.

6 Evaluation

6.1 Impact of Newly Identified Behaviors

This section first illustrates the main differences between CPA and CA regarding two key metrics (IPC and LLC occupancy) through the study of a example mix (#20). The mix consists of 1 critical (`xalancbmk_06`), 1 squanderer

application (milc_06), and 6 non-critical applications. This mix was chosen to help understand why the squanderer behavior is difficult to identify.

Figure 6 shows the IPC (before being restarted) and LLC occupancy (for the whole execution) achieved along the execution for each of the 8 applications in the example mix. Figure 6a and Figure 6b present the results for CA and CPA, respectively. Several observations can be made. Firstly, looking at the X-axis, whose length is bounded by the TT (that is, the execution time of the longest running application of the mix, xalancbmk_06), it can be seen that CPA improves TT significantly, nearly 20%. In this specific mix, CA achieves similar results to the Linux default NP policy. The main reason is that the behavior of squanderer applications is not correctly detected. Squanderer applications present a high MPKI_LLC. Therefore, CA confuses milc_06 with a critical application and allocates it into a partition with a greater amount of LLC. However, since squanderer applications present very little reuse, this additional space does not translate into performance gains. This behavior can be observed in the LLC occupancy graph of Figure 6a (right side), where milc_06 occupies a large fraction of the space during nearly the whole execution.

CPA detects and handles this newly identified behavior, so milc_06 is no longer allowed to occupy such a large LLC space. At the beginning of the execution (by second 5), milc_06 is identified as a squanderer application because it presents high MPKI_LLC and very low HPKI_LLC. This application shows the same behavior throughout its execution time so it remains in a separate CLOS with a reduced amount of LLC space. The space released by milc_06 is taken by xalancbmk_06, a critical application that benefits from additional cache space, and the remaining non-critical applications. Notice that gains in TT are due to xalancbmk_06, as a non-critical application is barely affected. Looking at the IPC graphs, we can see that even though milc_06 is assigned much less LLC space by CPA than by CA, so its IPC (and execution time) is unaffected. Compared to CA, CPA improves the IPC by nearly 4%.

6.2 Turnaround Time Evaluation

This section compares the turnaround time (TT) and average normalized turnaround time (ANTT) [17] of CPA, CA, and Dunn [3], a state-of-the-art partitioning policy.

Figure 7a plots the TT improvement (in percentage) of the studied approaches with respect to NP across 31 8-application workload mixes. CPA improves TT over 40% in three mixes over NP. On average this improvement is of 11%, which is slightly higher in Dunn (12%) and lower (6%) in CA. An interesting observation is that CPA improves TT considerably with respect to Dunn and CA in those mixes containing a squanderer application, e.g., mix #9 and mix #10. This is because neither Dunn nor CA consider block reuse and LLC occupancy; thus, they are not able to detect behaviors strongly related to these metrics such as those exhibited by squanderer applications; and, as explained above, a wrong classification of squanderer applications leads the system to performance losses. Finally, we would like to remark that CPA manages to reach improvements over 35% in four mixes.

While TT is primarily a user-oriented performance metric [18], it does not consider the performance losses of an application over isolated execution, which can lead to misleading conclusions. To deal with this fact, we study a complementary metric, ANTT, which should be analyzed alongside with TT. For a given mix, the ANTT is calculated as the average of the slowdown of the applications that make up the mix. The slowdown is calculated as the TT of the application in multi-program execution over the TT of the application when executed alone. Figure 7b shows the ANTT improvement (in percentage) achieved by the studied policies over NP for each workload mix executed. As observed, CPA shows the best results, reaching in three mixes improvements over 4%. In contrast, Dunn degrades this metric by more than 4% in four mixes since it is more aggressive policy that tries to benefit most those application showing highest slowdown (i.e., critical). An important observation is that in those mixes where Dunn outperforms CPA regarding TT, CPA is able to improve ANTT. For instance, CPA manages to reach a TT improvement higher than 50% in mix #18, where Dunn and CA improve by 58% and 20%, respectively. However, CPA outperforms the other two approaches in ANTT by up to 5%. This means that CPA successfully considers the applications' individual performance as well as the overall system performance and that the gains in the latter are not at the cost of damaging the performance of the co-running applications.

6.3 IPC Evaluation

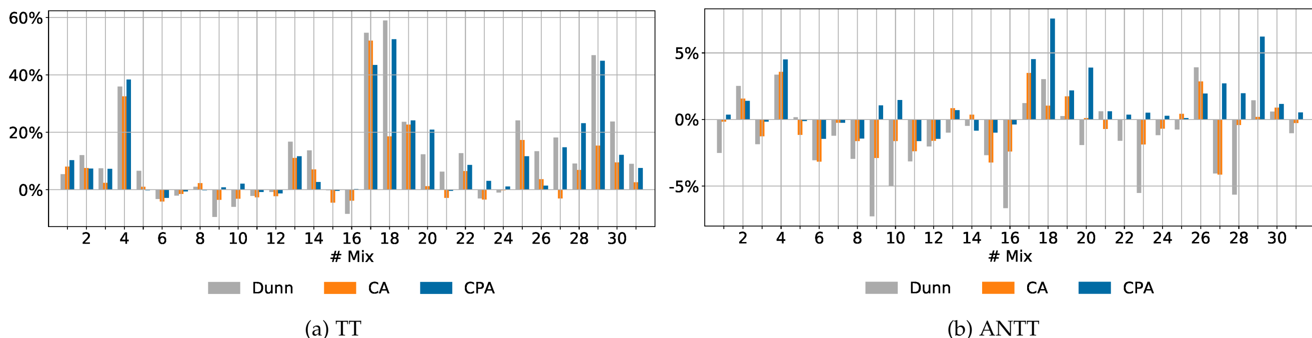
In addition to TT and ANTT, the system throughput is also evaluated in this work in terms of IPC. More specifically, the geometric mean of IPC is used since the raw-IPC or arithmetic mean can yield to misleading conclusions [19].

Figure 8 shows the improvement (in percentage) of the IPC geometric mean with respect to NP for each workload mix in the studied policies. Dunn allows the system to achieve a good system fairness, however, when problematic behaviors identified in this work are present in the mix, the IPC can drop. The IPC is difficult to sustain in partitioning approaches focused on multi-program workloads mainly because, to deal with system fairness or TT, these partitioning approaches seek to benefit those applications showing an atypical behavior at the expense of damaging the best performing ones. Nevertheless, in spite of this fact, the devised CPA approach properly addresses the newly identified behaviors, improving TT and ANTT while sustaining the IPC or even improving it over 3% in some mixes.

Overall, CPA (and CA but to a lesser extent) manages to maintain and even improve, in some cases, the IPC, whereas Dunn's IPC is worse than NP in most cases. Even though Dunn obtains, on average, a slightly better TT, this improvement should never be at the cost of degrading the system throughput. This principle was firstly drafted in CA and has been fully tackled in this work.

6.4 Comparing CPA with KPart

This section compares our proposal with KPart [5]. This approach groups applications into N clusters, this number ranges from 2 to the number of applications in the mix and each cluster is assigned to a cache partition, which has



(a) TT

(b) ANTT

Fig. 7: TT and ANTT improvement (in %) w.r.t. NP for each workload mix.

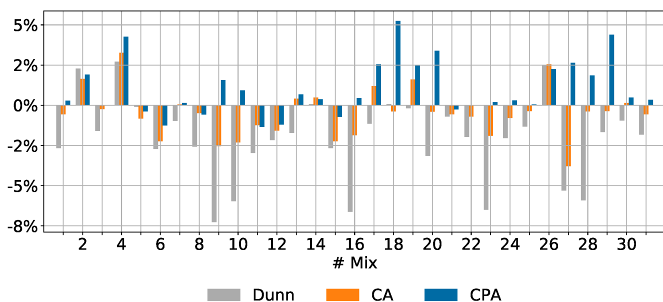


Fig. 8: IPC (geometric mean) improvement (in %) w.r.t. NP.

Application	KPart		CPA	
	Time (s)	IPC	Time (s)	IPC
sjeng_06	65.5	1.34	38.5	2.28
hmmer_06	60.1	1.45	41.0	2.14
nab_17	71.8	1.22	60.5	1.45
libquantum_06	38.3	2.29	61.5	1.43
tonto_06	41.6	2.10	63.5	1.38
soplex_06	109.3	0.80	65.0	1.35
sphinx3_06	61.1	1.43	85.5	1.03
mcf_06	211.9	0.41	123.0	0.71
TT/IPC(geomean)	**211.9**	**1.23**	**123.0**	**1.39**

TABLE 4: Execution time (s) and IPC the individual applications of mix #13, and TT and IPC (geomean) of the mix.

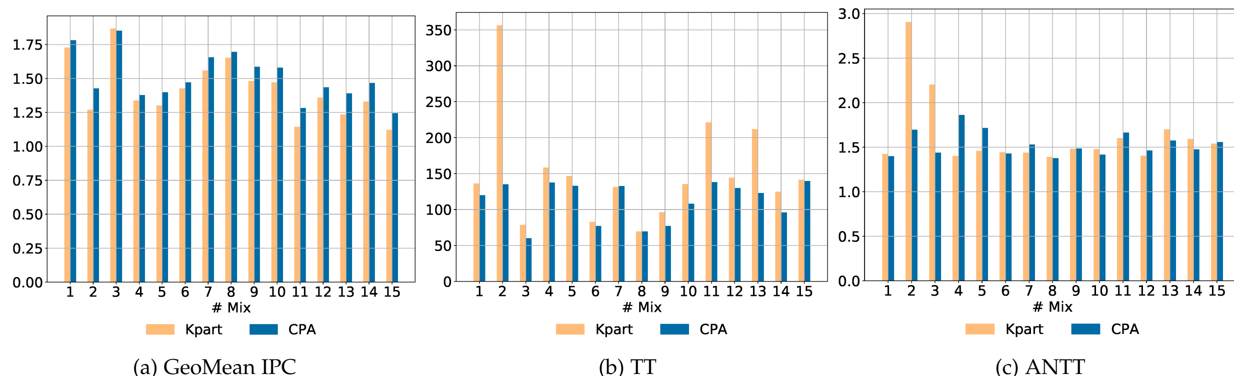
private ways. In order to run KPart in our experimental platform, we adapted the cache partitioning technique to work with a 20-way cache. We tried the same 31 mixes described in Section 5.1, but the KPart framework was only able to run 15 of them, as some SPEC benchmarks are composed of multiple binaries and supporting that requires important changes in the KPart architecture.

Results for IPC (geometric mean), TT and ANTT are shown in Figure 9. For fair comparison purposes, we also ran the application mixes in CPA until each application of the mix committed 200 B instructions, like in KPart. As observed, CPA outperforms KPart, on average, by 6% in IPC (geometric mean), 30% in TT and 6% in ANTT.

To provide further insight into why CPA outperforms KPart, we looked into those workloads where KPart presents a poor performance and found three main reasons: i) KPart allocates multiple applications to 1-way partitions, ii) KPart places problematic and critical applications together, and iii) CPA is triggered when it is really needed (phase changes), and does more precise changes.

For illustrative purposes, we present below a comparative study of mix #13, made up of 5 non-critical applications (`sjeng_06`, `hmmer_06`, `nab_17`, `libquantum_06`, `tonto_06`), 1 critical sensitive application (`soplex_06`), 1 critical medium application (`sphinx3_06`) and 1 bully application (`mcf_06`). Table 4 shows the results of the execution time and IPC obtained for each benchmark of this mix. As it can be observed, the performance (i.e., IPC) of some non-critical applications drops considerably compared to CPA. KPart allocates firstly these applications into the same 1-way CLOS, and then in a CLOS with very little space (1 way/application). Consequently, the partition behaves like a direct-mapped cache, which results into performance degradation. In contrast, CPA does not constrain non-critical applications to such a reduced space, thus it does not present this downside. Another important difference is that KPart is not able to correctly identify bully applications like `mcf_06`. In this example mix, KPart assigns firstly `mcf_06` too little space (1 way), which damages its performance considerably. In the following cache disposal (that is, clustering applications in CLOS and assigning them cache ways), it is placed together with `soplex_06` in a CLOS with a high number of ways (14). However, as shown in the characterization study, problematic applications (`mcf_06` in this example) tend to occupy and waste a high portion of their allocated LLC space. Therefore, placing them in the same cluster as critical applications results in lower performance, as they reduce the space available for the critical ones, without significant performance gains. In the last cache disposal, `mcf_06` is given too much space (9 private ways) that it is not able to use profitably. Notice that CPA mostly uses shared cache ways among CLOS which, as proved by [3], generally yields a better performance (see Section 4.3).

Another difference between KPart and CPA is how often the cache configuration is modified and the number of applications/partitions affected. In this mix, KPart performs a total of 3 cache disposals, compared to CPA which performs 10 cache configuration updates (following criteria in Table 3), and 23 cache adjustments (Step 5 in Algorithm 1). CPA performs more frequent and precise cache configuration updates, which adapts better to behavior changes of the applications.



(a) GeoMean IPC (b) TT (c) ANTT

Fig. 9: CPA versus KPart: IPC (geometric mean), TT and ANTT.

Consequently, in this example mix CPA improves the IPC (geomean) by 13% since it avoids low IPCs (2 out 8 applications present IPC below 1 in KPart). CPA also outperforms by 61% in TT. We carried out further experiments to estimate the impact of each of the analyzed aspects. We found that TT drops to 53% when phase detection is not applied, and to 27% when using 1-way partitions, proving that major performance gains come from the proposed strategy that identifies new specific cache behaviors and performs more precise cache configuration updates.

7 Related Work

The first approaches dealing with cache partitioning were implemented using simulation frameworks. Some approaches like UCP [8], ASM [20], Vantage [21] or PriSM [22] modify the eviction and insertion policies to partition the cache, hence they cannot be implemented in existing processors. Other approaches like the filter cache [23], split the cache in different structures to reduce the interference.

Recently, the research trend has changed, mainly due to the fact that some recent processors provide support to partition the cache, so nowadays the main focus is on implementing cache partitioning policies in commercial processors. Selfa *et al.* [3] cluster applications using the k-means algorithm and distribute cache ways between the groups, giving exponentially more space to the applications suffering more interference, in order to improve system fairness. El-Sayed *et al.* [5] also group applications into clusters, assigning them to different CLOS. While it manages to significantly improve throughput in selected workloads, it uses a detailed profiling, resulting in a much more complex algorithm than CPA. POCAT [24] uses *Intel Top-down Microarchitecture Analysis Method (TMAM)*, leveraging machine learning to predict applications' IPC for different cache sizes. This model also captures IPC behavior changes but it requires using a machine learning dataset, created by previously collecting TMAM metrics and IPC values of each application for each cache way setting. In contrast, CPA detects IPC changes online and it does not require the use of offline data. DCAPS [25] proposes a framework based on predictors that uses miss rate curves and LLC occupancy predictions. Their approach estimates the LLC occupancy from the number of misses incurred, something that can lead to wrong conclusions since there are applications that present a low number of LLC misses but a high number of memory accesses due to prefetches, so resulting in a high LLC occupancy. Contrary to this work, we measure the effective LLC occupancy to identify anomalous LLC behaviors that can drop the performance of the co-runners.

Finally, recent research works [9], [26], [27] have been published dealing with cache partitioning for cloud systems. These systems present workloads with other characteristics, like latency critical (LC) applications, where quality of service must be satisfied, which run jointly with best-effort applications to improve the resource utilization. The partitioning schemes, however, are much simpler (e.g., Heracles [9] only considers a partition for a single LC application).

8 Conclusions

This work has made two main contributions. Firstly, we have presented a detailed characterization of the LLC behavior of the applications. The study has analyzed the relationship between the system performance and the LLC performance. The analysis has resulted in two important findings: i) new cache behaviors like squanderer, bully, and medium have been found, and ii) applications may experience different LLC behaviors throughout their execution which match the IPC phases of the application. The main contribution of the study is that, if the newly identified LLC behaviors are not properly identified, the system performance can significantly drop. Secondly, this work has proposed the phase driven CPA approach, which relies on the results of the characterization analysis to balance the cache space. The proposed approach i) assigns a private CLOS to each individual application provided that it exhibits critical (sensitive or medium) or squanderer behavior, and ii) those non-critical applications that overuse LLC space are isolated in private and small partitions.

Experimental results show that CPA improves TT, over Linux default behavior about 40% in mixes including applications showing the newly identified behaviors. Compared to KPart, CPA outperforms on average TT by 30%, and IPC and ANTT by 6%. Compared to Dunn, CPA achieves a similar TT, but Dunn sacrifices system ANTT and IPC. CPA source code is available at http://hdl.handle.net/10251/143182.

Acknowledgments

This work has been partially supported by Ministerio de Ciencia, Innovación y Universidades and the European ERDF under Grant RTI2018-098156-B-C51, and Generalitat Valenciana under Grant AICO/2019/317.

References

[1] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke, "Ibm power9 processor architecture," *IEEE Micro*, vol. 37, no. 2, pp. 40–51, Mar 2017.

[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, Mar 2016.

[3] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez, "Application clustering policies to address system fairness with intel's cache allocation technology," in *Procedings of PACT*, 2017, pp. 194–205.

[4] L. Pons, V. Selfa, J. Sahuquillo, S. Petit, and J. Pons, "Improving system turnaround time with intel CAT by identifying LLC critical applications," in *Proceedings of Euro-Par*, 2018, pp. 603–615.

[5] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "Kpart: A hybrid cache partitioning-sharing technique for commodity multicores," in *Procedings of HPCA*, 2018.

[6] *Standard Performance Evaluation Corporation, SPEC CPU 2006: http://spec.org/cpu2006*.

[7] *Standard Performance Evaluation Corporation, SPEC CPU 2017: http://spec.org/cpu2017*.

[8] 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 MICRO*, 2006, pp. 423–432.

[9] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving Resource Efficiency at Scale," in *Proceedings of ISCA*, 2015, pp. 450–462.

[10] H. Zhu and M. Erez, "Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems," in *Proceedings of ASPLOS*, 2016, pp. 33–47.

[11] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *Proceedings of MICRO-36.*, Dec 2003, pp. 217–227.

[12] G. L. T. Chetsa, L. Lefevre, J.-M. Pierson, P. Stolf, and G. da Costa, "A user friendly phase detection methodology for hpc systems' analysis," in *Proceedings of GREENCOM-ITHINGS-CPSCOM*, 2013, pp. 118–125.

[13] A. Sembrant, D. Eklov, and E. Hagersten, "Efficient software-based online phase classification," in *Proceedings of IISWC*, Nov 2011, pp. 104–115.

[14] X. Liao, R. Guo, D. Yu, H. Jin, and L. Lin, "A phase behavior aware dynamic cache partitioning scheme for cmps," *International Journal of Parallel Programming*, vol. 44, pp. 68–86, 02 2016.

[15] J. Miller, "Short report: Reaction time analysis with outlier exclusion: Bias varies with sample size," *Journal of Experimental Psychology*, vol. 43, no. 4, pp. 907–912, 1991.

[16] I. M. T. Gleixner, "Performance counters for linux," 2009.

[17] K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in smt processors," 02 2001, pp. 164 – 171.

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

[19] 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.

[20] 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 MICRO*, 2015, pp. 62–75.

[21] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-grain Cache Partitioning," in *Proceedings of ISCA*, 2011, pp. 57–68.

[22] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic Shared Cache Management (PriSM)," in *Proceedings of ISCA*, 2012, pp. 428–439.

[23] J. Sahuquillo and A. Pont, "The filter cache: A run-time cache management approach1," in *25th EUROMICRO '99 Conference*.

[24] Y. Kim, A. More, E. Shriver, and T. Rosing, "Application performance prediction and optimization under cache allocation technology," in *2019 Design, Automation Test in Europe Conference Exhibition*, March 2019, pp. 1285–1288.

[25] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "Dcaps: Dynamic cache allocation with partial sharing," in *Proceedings of EuroSys*. ACM, 2018, pp. 13:1–13:15.

[26] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of ASPLOS*, 2019, pp. 107–120.

[27] J. Park, S. Park, and W. Baek, "Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers," in *Proceedings of EuroSys*, 2019, pp. 10:1–10:16.

Lucia Pons received the BS and MS degrees in computer engineering from the Universitat Politècnica de València (UPV), Spain, in 2018 and 2019, respectively. She is currently working towards a PhD degree at the Department of Computer Engineering (DISCA) of the same university. Her PhD research focuses on cache partitioning approaches and efficient use of shared resources in multi-core.

Julio Sahuquillo (M'04) received the BS, MS, and PhD degrees from the UPV, Spain, all in computer engineering. He is a Full Professor with the DISCA department at the UPV. He has taught several courses on computer organization and architecture. He has authored over 150 refereed conference and journal papers. His current research interests include multicore processors, memory hierarchy design, GPU architecture, and resource management.

Vicent Selfa received the BS, MS and PhD degrees in computer engineering from the UPV, Spain. He collaborates with the Parallel Architecture Group (GAP) of the Universitat Politècnica de València. His research interests include fairness-aware resource partitioning policies, cache prefetching, and chip multiprocessor architectures.

Salvador Petit (M'07) received the PhD degree in computer engineering from the UPV. Since 2009, he has been an Associate Professor with the DISCA department, where he has taught several courses on computer organization. He has authored over 100 refereed conference and journal papers. His current research interests include multi-core processors, memory hierarchy design, GPU architecture, and resource management.

Julio Pons received the BS, MS, and PhD degrees from the UPV, Spain, all in computer engineering. He is an Associate Professor with the DISCA department. He has taught several courses on computer organization and operating systems. His current research interests include multi-core processors, memory hierarchy design, cache sharing and cloud computing.