Improving System Turnaround Time with Intel CAT by Identifying LLC Critical Applications

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

Department of Computer Engineering, Universitat Politècnica de València, Spain
lupones@inf.upv.es

Abstract. Resource sharing is a major concern in current multicore processors. Among the shared system resources, the Last Level Cache (LLC) is one of the most critical, since destructive interference between applications accessing it implies more off-chip accesses to main memory, which incur long latencies that can severely impact the overall system performance. To help alleviate this issue, current processors implement huge LLCs, but even so, inter-application interference can harm the performance of a subset of the running applications when executing multiprogram workloads. For this reason, recent Intel processors feature *Cache Allocation Technologies* (CAT) to partition the cache and assign subsets of cache ways to groups of applications. This paper proposes the *Critical-Aware* (CA) LLC partitioning approach, which leverages CAT and improves the performance of multiprogram workloads, by identifying and protecting the applications whose performance is more damaged by LLC sharing. Experimental results show that CA improves turnaround time on average by 15%, and up to 40% compared to a baseline system without partitioning.

1 Introduction

Recent processors implement huge Last Level Caches (LLC) and prefetching mechanisms to hide long main memory access latencies. LLC caches are typically shared among all applications running in the cores. The LLC is sized according to the processor core count, and typically takes a few MBs (e.g. 1 to 4) per core, resulting in an overall cache capacity in the order of tens of MBs. This capacity is even higher in processors using 3D stacking technologies [1].

Cache sharing yields important problems to the system from a performance perspective. These problems appear due to inter-application interferences at the shared cache, making the system become unpredictable. As a consequence, issues like system throughput [2–4], and fairness [5–8] have been addressed in previous research.

Most research work on cache sharing during the last decade has been developed using simulators because hardware in commercial processors did not support it. Fortunately, some processors manufacturers like Intel and ARM have recently deployed technologies that allow distributing LLC cache ways among

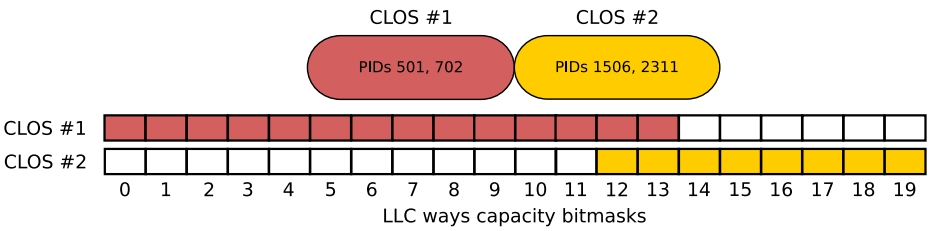


Fig. 1: Intel CAT example with PIDs associated to two CLOSes.

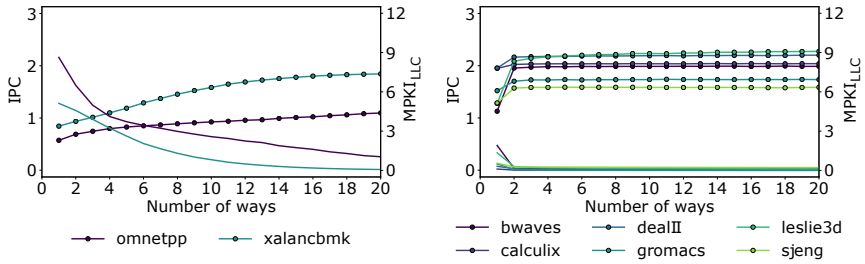
co-running applications. For instance, Intel has deployed *Cache Allocation Technology* (CAT) which is being delivered in some recent processors.

These technologies provide a hardware-software interface, which allows the software to take decisions about the cache distribution. This means that a cache sharing approach can be more elaborated than if it was entirely implemented in hardware.

The main focus of this paper is to reduce the **turnaround time** of a set of applications running concurrently (mix). This time is defined as the elapsed time between the mix's execution start and the instant the last application of the mix finishes. Turnaround time is especially important in batch based systems, since improving this performance metric allows the system to transit to a low power state and helps energy savings. Eyerman and Eeckhout [9] claim that program turnaround time in general-purpose systems and interactive environments should be considered one of the primary performance criteria.

In this work we characterize the LLC behavior of applications in isolated execution, and their dynamic behavior when they are executed with other co-runners. This study revealed that some applications, *namely critical*, should be protected (i.e. exposed to less inter-application interference) by assigning them exclusive cache ways. On the contrary, other applications, *namely non-critical*, present *fewer* space requirements, hence suffering less from inter-application interference.

Based on this characterization study we propose the Critical-Aware (CA) algorithm, a simple and low-overhead LLC partitioning approach, which is the main contribution of this work. The fact that it is simple and has a minimum cost, is key to adapt to different execution phases, as fast and frequent decisions are needed to be taken. CA approach, by design, divides the cache in two partitions (one for critical and one for non-critical applications) and dynamically assigns or removes ways to each group, allowing some degree of way sharing between them. Experimental results show that CA improves turnaround time by up to 40% and system unfairness over 55%, compared to execution under a baseline system without partitioning.



(a) Critical applications (b) Non-critical applications

Fig. 2: Example of behavior of critical and non-critical applications in a workload. Legend: marked lines (with circles) represent IPC and solid lines MPKI_{LLC}.

2 Intel Cache Allocation Technology

Intel Cache Allocation Technology allows assigning a given amount of LLC cache ways to a set of applications. This technology is available in limited models of processors of the Xeon E5 2600 v3 family and in all processors of the Xeon E5 v4 family. CAT allocates PIDs (Processors Identifiers) or logical cores to Classes Of Service (CLOSes). In the most recent processors, the maximum number of CLOSes is 16. For each CLOS, the user has to specify i) the subset of ways that can be written, and ii) which applications or logical cores belong to the CLOS. The cache ways that can be written by the applications belonging to a CLOS are defined with a *capacity bitmask* (CBM). Cache ways are not necessarily private to a CLOS, as they can be shared with other CLOSes by overlapping the CBMs.

Figure 1 shows an example for a possible CAT configuration using 2 CLOSes and a cache with 20 ways. In this example, we assign PIDs to the CLOSes instead of logical cores, a feature available starting with Linux 4.10. Note that CLOS #0 is special, in the sense that is the one used by default by applications that have not been assigned to a particular CLOS. Also note that due to hardware limitations, the ways assigned to a given CLOS have to be consecutive.

3 Application Characterization

In order to design the partitioning algorithm, we performed a characterization of the applications of the SPEC2006 [10] benchmark suite from the LLC perspective, illustrating the relationship between cache space and overall performance. To this end, we executed each application with a cache space from 1 to 20 ways. After analyzing the results, we concluded that applications can be divided in two main categories, according to the impact that the cache space has on their performance: cache critical applications and non-cache critical applications. The non-cache critical applications do not show significant performance gains as they have more cache space for a number of ways greater than one. On the other hand, cache critical applications experience important performance gains as the number of ways increases. However, these performance gains diminish as the number

Critical Applications
omnetpp, soplex, sphinx3, xalancbmk, lbm, mcf
Non-critical Applications
astar, bwaves, bzip2, cactusADM, calculix,dealII, gamess, gobmk, gromacs, h264ref, hmmer, leslie3d, libquantum, namd, povray, sjeng, tonto, wrf, zeusmp

Table 1: Categorization of SPEC2006 applications.

of assigned cache ways reaches a given threshold (e.g. 10 or 12). Table 1 classifies the studied applications in cache critical and non-cache critical.

Figure 2 illustrates both behaviors, showing how the Instructions Per Cycle, IPC, (dotted line, Y axis scale on the left) and the LLC Misses Per Kilo Instructions, MPKI_{LLC}, (plain line, Y axis scale on the right) change as the number of ways increases for eight applications running alone. Figure 2a and Figure 2b show the results for critical and non-critical applications, respectively. Notice that both IPC and the MPKI_{LLC} are almost constant for non-cache critical applications, varying only when just one way is available. The reason of this divergent behavior with one way is that this configuration behaves like a direct-mapped cache, which presents a high number of conflict misses. Notice also that the MPKI_{LLC} in these applications is always below 0.5. In contrast, critical applications, like `omnetpp` and `xalancbmk` exhibit a much higher MPKI_{LLC} that decreases with the number of assigned cache ways, which results in IPC improvements.

We also found that the huge LLC space (20MB) in our experimental platform, is completely occupied, or close to being so, by any of the studied applications when running in isolation, regardless being cache critical or not. This is due to the huge working set that studied applications use considering the entire execution. Results in Figure 2b, however, show that the *live working set* of non-critical applications for a specific interval of time fits in two cache ways.

This analysis has yielded key observations that are the pillars on which CA relies: i) assigning two cache ways to non-critical applications suffices for performance, ii) non-critical applications should have a limited cache space; otherwise, they will occupy precious cache space that, on the one hand will not result in individual performance gains, and on the other hand, will prevent critical applications from using it, indirectly harming their performance, and iii) critical applications need to be protected by assigning them a significant fraction of the cache ways to preserve their performance. To provide further insights into these claims we performed a dynamic analysis, discussed below.

Dynamic characterization To study the impact of sharing the cache with other co-runners in multicore execution, we characterized a large set of randomly generated mixes. For illustrative purposes, we used mix #23, composed of the eight previously studied applications, when running together in a baseline system without partitioning.

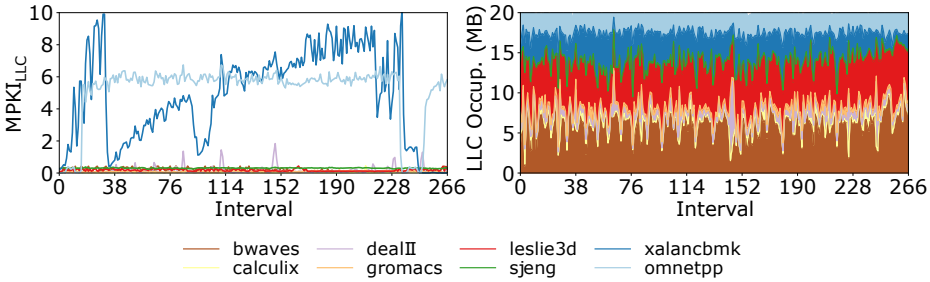


Fig. 3: LLC dynamic occupancy and MPKI_{LLC} in mix #23 without partitioning.

To check if a non-partitioning scheme meets the demands of individual applications, we analyzed the LLC occupancy per application and its MPKI_{LLC}. It can be observed in Figure 3 that non-critical applications like `bwaves` occupy almost all the time between five and seven ways (which represents between one quarter and one third of the total cache space) in spite of, as shown in the static analysis, barely needing two cache ways to achieve its maximum performance. On the contrary, `xalancbmk` which has been identified as a critical application, presents a much lower cache occupancy than some of the non-critical ones. As a result, its MPKI_{LLC} severely rises (Figure 3), turning into important IPC drops.

This example supports our three claims mentioned above, which are the basis of which the CA approach relies.

4 The Critical-Aware Partitioning Approach

This section presents the Critical-Aware Partitioning Approach (CA). The general idea behind our proposal is to dynamically determine which applications present a cache critical behavior and divide the LLC into two partitions, one larger for the cache critical applications and the other, smaller, for the rest of applications. Since not all the cache critical applications show the *same criticality*, CA tries to further refine the partitions, dynamically readjusting their size, until the number of applications showing critical behavior changes, and the process is restarted. CA consists of three main phases: cache warm-up and application classification, partition allocation, and dynamic readjusting of cache ways, which are discussed next.

4.1 Cache Warm-Up and Application Classification

The first step in this phase is the cache `warm-up` phase (e.g. 10 intervals of 0.5 seconds), at the beginning of the execution, where no data is collected and therefore no action is taken. After that, the algorithm enters the `reset` state, in which, using the hardware counters `mem_load_uops_retired.l3_miss` and `instructions`, the MPKI_{LLC} for all the applications is computed.

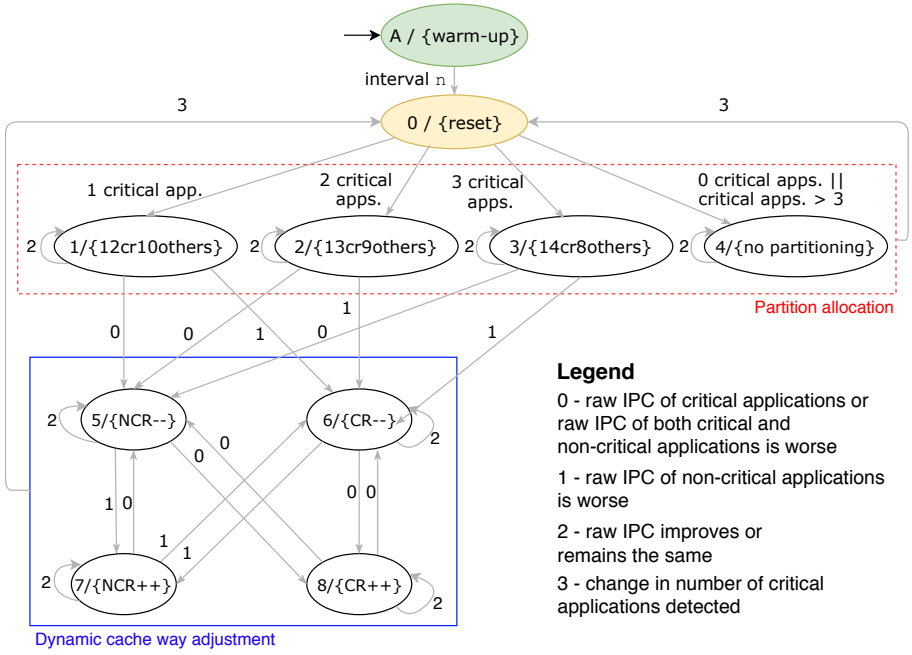


Fig. 4: State diagram of CA algorithm. Acronyms: NCR stands for the number of ways assigned to the CLOS with non-critical applications and CR stands for the number of ways assigned to the CLOS with critical applications.

As concluded in Section 3, critical applications experience a much higher MPKI_{LLC} than non-critical ones. Therefore, considering that there are less critical than non-critical applications (less than 25% in the SPEC2006 suite), it is expected that critical applications are outliers, with respect to the general MPKI_{LLC} trend. So, following this rationale, the algorithm computes the rolling mean of the MPKI_{LLC} for each application, and then, using the Miller's criterion [11], all the applications with a MPKI_{LLC} greater than three standard deviations from the average of the total rolling MPKI_{LLC}, are classified as critical. We have also tested other methods to detect outliers, like the Median Absolute Deviation (MAD) [12], but the former approach achieved better results for our purposes.

4.2 Partition Allocation

After the applications have been classified, the algorithm creates two partitions, whose sizes depend on the number of critical applications detected, and assigns applications to one or another, depending on their criticality. This stage is marked in red in Figure 4. The used partition layouts were empirically determined based on a depth and thorough study of static configurations, evaluating

Configuration name	# of Critical Appl.	CLOS 1 ways	CLOS 2 ways
12cr10others	1	12	10
13cr9others	2	13	9
14cr8others	3	14	8
no_partitioning	0 or more than 3	20	20

Table 2: Initial cache mask configurations.

application mixes with different numbers of critical applications, listed in Table 2. These partitioning layouts are based on the fact that non-critical applications need an average of 2 cache ways to sustain their performance. The next step of the algorithm (*Dynamic cache way adjustment* box in Figure 4) is devoted to fine-tune the partitions, iteratively varying their size. While CA could use this mechanism right from the start to dynamically converge to the best configuration, predefined configurations (*Partition Allocation* box in Figure 4) are used as a starting point to speed up convergence towards it.

Selfa *et al.* [5] proved that a design with CLOSes having a fraction of the cache ways shared with other CLOSes results in a better cache utilization. CA takes this claim into account and grants the CLOS hosting the critical applications a high number of exclusive cache ways, and a small fraction of shared ways. The more critical applications detected, the more cache ways are assigned to its CLOS, except for the workload mixes where a majority or no critical applications are detected. In these cases, the cache configuration remains untouched (all the applications have 20 ways), as limiting the space of a dominant number of critical applications and placing them together does not improve performance.

4.3 Dynamic Adjusting of Partitions

Since there is not an optimal cache configuration that perfectly suits all the application mixes, CA readjusts the initial configuration dynamically, depending on its response to the changes performed. To check how good a change in the actual partitioning is, CA needs to estimate what would have been the performance if such a change was not made. A simple approach is to assume that the IPC in the next measured interval will remain similar as in the previous one. This provided us estimates with around 4% Mean Square Error. Other methods, like using an Exponentially Weighted Moving Average and ARIMA models were also evaluated, but the reduction in the prediction error did not compensate the increase in complexity.

The dynamic adjusting (*Dynamic cache way adjustment* box in Figure 4) has 4 states, labeled `5`, `6`, `7` and `8`. States `5` and `8` decrease and increase, respectively, the number of ways of the partition of non-critical applications. On the other hand, states `6` and `7` decrease and increase, respectively, the number of ways of the partition that has the critical applications mapped. Transitions can happen due to four reasons, numbered as they appear in the figure:

0. The raw IPC of critical applications or the raw IPC of both groups is worse than the estimated IPC.
1. The raw IPC of non-critical applications is worse than the estimated IPC.
2. The raw IPC improves the estimated IPC. This causes a transition to the same state, and no action is taken for 5 intervals.
3. The number of critical applications has changed. We perform a reset in the partitioning, and depending on the number of critical applications detected, a new partitioning layout will be applied.

In the case where the raw IPC is worse than the prediction, a deeper study of the IPC is carried out to analyze where exactly is the source of performance loss. When the raw IPC of either non-critical or critical applications is lower than estimated, then a shared cache way becomes exclusive to the CLOS holding the affected applications. In case the total IPC of both CLOSes is worse than estimated, the same actions are taken as if only the raw IPC of the critical applications was worse. Giving more cache space to critical applications will result in a higher increase in performance than if it is given to the non-critical applications (see Section 3). If any action taken results in a performance loss of the other CLOS, the action is reverted, i.e. the exclusive way is transformed back into a shared way.

Since applications have a non-uniform behavior during their execution, the number of applications detected as outliers can vary considerably along the execution, which can result in a high number of *resets* (i.e. transiting to the *reset* state), and consequently, in a reduction of the system throughput. To deal with this fact, if in a given interval an application is not detected as an outlier but it has been critical during more than 50% of the execution time, it is again considered as critical.

5 Experimental Framework

The experiments have been conducted in a machine with an Intel Xeon E5-2620 v4 processor, with 8 SMT cores running at 2.20GHz. It has a 20-way LLC of size 20MB (1MB / way) that supports CAT, having available 16 CLOSes. Only CLOS #1 and #2 were used in the experiments. We deliberately skipped CLOS #0 since it is the default CLOS, as explained in Section 2, and using it affects *all* the processes in the system, including the kernel processes.

To conduct the experiments, we have developed a framework that measures performance using a library based on *Linux perf* [13], and partitions the cache using the primitives provided by the Linux kernel. The results presented in this paper have been obtained using a vanilla Linux 4.11. Our framework samples the performance counters every half a second. The information gathered is used to guide the partitioning policy. When no partitioning is applied, the overhead of collecting data is around 1%. With our proposal, which collects data and executes the partitioning algorithm, the overhead is 1.3%. This means that the overhead caused by the execution of CA is minimal.

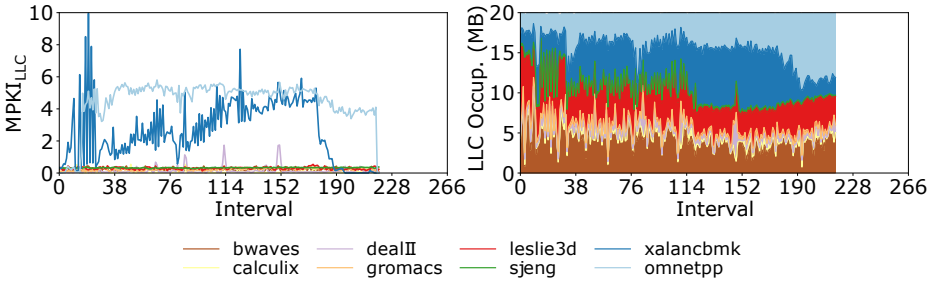


Fig. 5: LLC dynamic occupancy and MPKI$_{LLC}$ in mix #23 with CA algorithm.

The workload mixes were randomly generated using 25 applications from the SPEC2006 [10] benchmark suite. To guide their design, we used the insights presented in Section 3, which showed that the number of critical applications is much lower than the number of non-critical ones. Taking this observation into account, we generated 34 application mixes, each one with 8 applications, with the applications randomly selected from the categories shown in Table 1. We generated 17 mixes (mixes 1 to 17) with 1 critical application, 10 mixes (mixes 18 to 27) with 2 critical applications, and 7 mixes (mixes 29 to 34) with 3 critical applications.

The experimental methodology followed was to execute each workload until all the applications completed the same number of instructions they execute when running alone for 60 seconds. When an application reached this limit, and it is not the last one to reach such limit, it is restarted, but only the results of the first run were considered. Each experiment was repeated 3 times, and the average was derived.

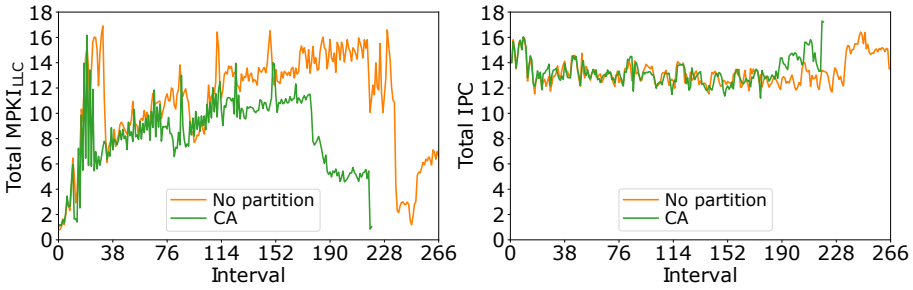
6 Evaluation

This section evaluates the performance and fairness of the CA approach, against a baseline system where no partitioning is done, referred to as *No partition*; but first, we illustrate how CA achieves its objectives through a case study.

6.1 LLC Dynamic Occupancy and MPKI with the CA approach

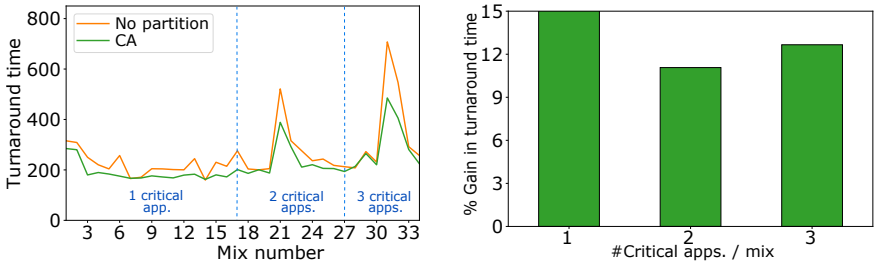
This section illustrates how the proposal properly distributes the cache space, by assigning it according to the type of application. For this purpose, we use the same mix example (i.e. mix #23) as in Section 3.

Figure 5 shows how the LLC occupancy and MPKI$_{LLC}$ of the 8 benchmarks of mix #23 evolves along time under the CA approach. Compared to non-partitioned system (see Figure 3), it can be observed that non-critical applications (e.g. `bwaves`) are allowed to use less space, while critical applications (e.g. `xalancbmk`) are assigned more cache space, which confirms that CA properly addresses the design objectives. Notice that both compared figures have the



(a) MPKI-L3 (b) IPC

Fig. 6: CA vs Linux: Dynamic cumulative MPKI$_{LLC}$ and raw IPC in mix #23.



(a) Individual workload mixes. (b) Grouped workload mixes.

Fig. 7: Turnaround time of CA vs Linux.

same number of intervals (X axis), however, the mix execution ends, under CA , around interval 205, which means that the gain in turnaround time is about 22%. This is achieved by significantly reducing the MPKI$_{LLC}$, clearly appreciated by comparing the results of each individual application applying CA (see Figure 5) with those achieved with Linux reference system (see Figure 3). Under Linux, critical applications like `xalancbmk` suffer a high MPKI$_{LLC}$, exceeding 6 around half of the execution time.

The previous observations can also be appreciated by presenting the aggregated MPKI$_{LLC}$ and IPC values of all the benchmarks that make up the mix, for both the reference system and CA . This can be seen in Figure 6. As can be observed, the aggregated MPKI$_{LLC}$ is much better with CA than with a non-partitioned system, which translates into ending the execution much before, hence improving system throughput. Regarding raw IPC, it can be appreciated that CA achieves similar or slightly better (around intervals 190 to 210) raw IPC than the baseline system.

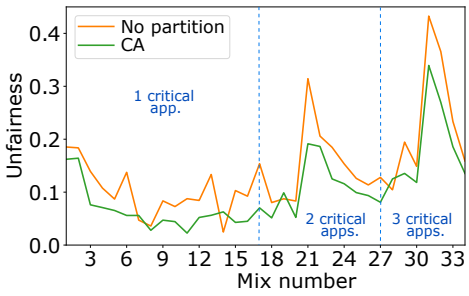


Fig. 8: Comparison of average unfairness value of the workload mixes.

6.2 Performance and Fairness

Performance has been analyzed in terms of turnaround time, which is the main focus of the CA approach. Figure 7 shows the turnaround times (in intervals of half a second) of both CA and the reference system across the 34 studied mixes. As observed, CA reduces the turnaround time of the mixes that have only one critical application (mixes 1 to 17), on average, by 15%. For mixes with two critical applications (mixes 18 to 27) the reduction is by around 11.5% and for mixes with 3 critical applications (mixes 28 to 24) it decreased 13.5%. Note that for some of the mixes, the turnround time is reduced by more than 40%.

This turnaround time reductions are achieved by allowing slower applications (i.e. critical) to run faster and by slightly slowing down non-critical applications. As a result, the execution time of fast and slow applications becomes closer, hence improving system fairness. A comparison of unfairness values for each workload using both policies is presented in Figure 8. Unfairness is calculated as the standard deviation across all progresses of each individual application over the average progress, which in turn is calculated as the the total time when executing alone over the total time taken in multiprogram execution [14]. CA reduces unfairness, on average, by more than 55%, being this value even larger on mixes with one critical application, which is the most common case. In these mixes the percentage gain rises to 75%.

7 Related Work

Heracles [15] and Dirigent [16] focus on improving the performance of latency sensitive applications, determined *a priori*, by ensuring the other *batch* applications do not interfere with them. While in this work we also use the concept of *critical application*, we determine them dynamically, by measuring their cache behavior. Ginseng [17] partitions the LLC into isolated partitions using a marked-driven auction system. It is focused in environments like the cloud, where each guest can bid based on the amount of resources it wants to use. Selfa *et al.* [5] cluster applications using the k-means algorithm and distribute cache ways between the groups, giving exponentially more space to the applications

suffering more interferences, in order to improve system fairness. El-Sayed *et al.* [4] also group applications into clusters, assigning them to different CLOSes. While it manages to significantly improve throughput in selected workloads, it uses a detailed profiling, resulting in a much more complex algorithm than CA.

Some approaches like UCP [2], ASM [18], Vantage [19] or PriSM [3] modify the eviction and insertion policies to partition the cache, hence they cannot be implemented in existing processors. Other approaches like the filter cache [20], split the cache in different structures to reduce inter-application interferences.

8 Conclusions

Recent cache partitioning approaches implemented in commercial processors have been shown to be effective addressing fairness and throughput.

This paper has characterized the static and dynamic cache behavior when multiple applications compete among them for cache space, finding that some applications need few ways to achieve their maximum performance while some others require a higher number of ways. Therefore, they should be protected to avoid performance losses, by allocating them in a separate cache partition with a greater amount of cache space.

Based on this study, we have proposed CA, a simple and effective cache partitioning approach that significantly improves the turnaround time of workload mixes. Experimental results show that CA improves turnaround time up to 40% and system unfairness over 55%, compared to a baseline system without partitioning.

Acknowledgment

This work was supported by the Spanish Ministerio de Economía y Competitividad (MINECO) and Plan E funds, under grants TIN2015-66972-C5-1-R and TIN2017-92139-EXP. It was also supported by the ExaNest project, with funds from the European Union Horizon 2020 project, with grant agreement No 671553.

References

1. Sodani, A., Gramunt, R., Corbal, J., Kim, H.S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., Liu, Y.C.: Knights landing: Second-generation intel xeon phi product. IEEE Micro **36**(2) (Mar 2016) 34–46
2. Qureshi, M.K., Patt, Y.N.: Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In: Proceedings of MICRO. (2006) 423–432
3. Manikantan, R., Rajan, K., Govindarajan, R.: Probabilistic Shared Cache Management (PriSM). In: Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA). (2012) 428–439

4. El-Sayed, N., Mukkara, A., Tsai, P.A., Kasture, H., Ma, X., Sanchez, D.: Kpart: A hybrid cache partitioning-sharing technique for commodity multicores. In: Procedings of HPCA. (2018)
5. Selfa, V., Sahuquillo, J., Eeckhout, L., Petit, S., Gómez, M.E.: Application clustering policies to address system fairness with intel's cache allocation technology. In: Procedings of PACT. (2017) 194–205
6. Feliu, J., Sahuquillo, J., Petit, S., Duato, J.: Addressing Fairness in SMT Multicores with a Progress-Aware Scheduler. In: Proceedings of IPDPS. (2015) 187–196
7. Van Craeynest, K., Akram, S., Heirman, W., Jaleel, A., Eeckhout, L.: Fairness-aware Scheduling on single-ISA Heterogeneous Multi-cores. In: Proceedings of PACT. (2013) 177–188
8. Wu, C., Li, J., Xu, D., Yew, P.C., Li, J., Wang, Z.: FPS: A Fair-Progress Process Scheduling Policy on Shared-Memory Multiprocessors. Journal on Transactions on Parallel and Distributed Systems **26**(2) (2015) 444–454
9. Eyerman, S., Eeckhout, L.: System-level performance metrics for multiprogram workloads. IEEE Micro **28**(3) (May 2008) 42–53
10. Henning, J.L.: Spec cpu2006 benchmark descriptions. Comput. Archit. News **34**(4) (September 2006) 1–17
11. Miller, J.: Short report: Reaction time analysis with outlier exclusion: Bias varies with sample size. Journal of Experimental Psychology **43**(4) (1991) 907–912
12. Leys, C., Ley, C., Klein, O., Bernard, P., Licata, L.: Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. Journal of Experimental Social Psychology **49**(4) (2013) 764 – 766
13. T. Gleixner, I.M.: Performance counters for linux. (2009)
14. Van Craeynest, K., Akram, S., Heirman, W., Jaleel, A., Eeckhout, L.: Fairness-aware scheduling on single-isa heterogeneous multi-cores. In: Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques. PACT '13, Piscataway, NJ, USA, IEEE Press (2013) 177–188
15. Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., Kozyrakis, C.: Heracles: Improving Resource Efficiency at Scale. In: Proceedings of ISCA. (2015) 450–462
16. Zhu, H., Erez, M.: Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems. In: Proceedings of ASPLOS. (2016) 33–47
17. Funaro, L., Ben-Yehuda, O.A., Schuster, A.: Ginseng: Market-Driven LLC Allocation. In: Proceedings of USENIX. (2016) 295–308
18. Subramanian, L., Seshadri, V., Ghosh, A., Khan, S., Mutlu, O.: The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory. In: Proceedings of MICRO. (2015) 62–75
19. Sanchez, D., Kozyrakis, C.: Vantage: Scalable and Efficient Fine-grain Cache Partitioning. In: Proceedings of ISCA. (2011) 57–68
20. Sahuquillo, J., Pont, A.: The filter cache: A run-time cache management approach1. In: 25th EUROMICRO '99 Conference, 1999