The Journal of Supercomputing (2023) 79:10252–10276
https://doi.org/10.1007/s11227-023-05070-0

BALANCER: bandwidth allocation and cache partitioning for multicore processors

Agustín Navarro-Torres[1] · Jesús Alastruey-Benedé[1] · Pablo Ibáñez[1] · Víctor Viñals-Yúfera[1]

Accepted: 16 January 2023 / Published online: 4 February 2023
© The Author(s) 2023

Abstract

The management of shared resources in multicore processors is an open problem due to the continuous evolution of these systems. The trend toward increasing the number of cores and organizing them in clusters sets out new challenges not considered in previous works. In this paper, we characterize the use of the shared cache and memory bandwidth of an AMD Rome processor executing multiprogrammed workloads and propose several mechanisms that control the use of these resources to improve the system performance and fairness. Our control mechanisms require no hardware or operating system modifications. We evaluate Balancer on a real system running SPEC CPU2006 and CPU2017 applications. Balancer tuned for performance shows an average increase of 7.1% in system performance and an unfairness reduction of 18.6% with respect to a system without any control mechanism. Balancer tuned for fairness decreases the performance by 1.3% in exchange for a 64.5% reduction of unfairness.

Keywords Multicore processors · Memory hierarchy · Memory bandwidth · Shared last-level cache (LLC) · Cache occupancy · AMD Rome

✉ Agustín Navarro-Torres
agusnt@unizar.es

Jesús Alastruey-Benedé
jalastru@unizar.es

Pablo Ibáñez
imarin@unizar.es

Víctor Viñals-Yúfera
victor@unizar.es

[1] Dpto. Informática e Ingeniería de Sistemas - I3A, Universidad de Zaragoza, Zaragoza 50018, Spain

1 Introduction

Processors have an increasing number of cores and execution threads. For instance, AMD integrates up to 64 cores capable of executing 128 threads in the Rome and Milan microarchitectures [1], and the IBM Power10 processor can execute up to 120 threads, either with 15 SMT8 cores or 30 SMT4 cores [2]. This trend places greater pressure on shared resources, whose capacities, especially the last-level cache (LLC) and off-chip memory bandwidth, should be scaled and/or shared in the best possible way.

Shared resources can be managed by imposing limits on their use by one or more threads. In this way, each thread can only use its allocated quota, thus reducing interference with other threads. In recent years, commercial processors such as Intel Xeon [3], ARM ThunderX [4], or AMD EPYC [5, 6], have included hardware support for users to control the allocation of both LLC space and memory bandwidth to processor threads.

Hardware support for LLC space management has given rise to many proposals pursuing one or more goals, such as improving system performance [4, 7–15] or fairness [16, 17], facilitating server consolidation and/or ensuring quality of service (QoS) [11, 18–23], isolating tasks to decrease the worst case execution time (WCET) [24], decreasing the Turnaround Time [25], or seeking Social Welfare [26].

However, since processor support for memory traffic management is more recent, work on this topic is scarcer. Specifically, we are aware of only three proposals that control the memory bandwidth allocation to improve performance [11] or to support server consolidation [18, 27].

As shown in Table 1, almost all previous works focus on Intel memory hierarchies, where the LLC is shared among all cores, but there are no proposals specifically designed for a clustered LLC organization, such as the one used by AMD in its contemporary processors [5, 6]. We believe that this type of clustered organization is a promising trend, since increasing the number of processor cores makes a cache shared by all cores more inefficient in terms of access latency and interconnect network traffic. In fact, clustered organizations can be found in recent high-performance processors such as AMD EPYC [5, 6], IBM Power 10 [2], Fujitsu A64FX [28]. Moreover, a clustered design allows the desired growth in the number of transistors to be achieved economically by integrating a set of separately manufactured dies (chiplets in AMD terminology [29]) into a passive module.

Focusing on the AMD EPYC, the LLC is shared among only a few cores and the main memory access links form a hierarchical tree that terminates in several DRAM channels (details in Sect. 2). In this case, the management of LLC and memory bandwidth may become more complex, as the potential occurrence of local, intermediate, or global bottlenecks may have to be taken into account. Moreover, the application behavior in clustered hierarchies, especially when all the cores are running threads, has also not been studied. Not knowing this behavior makes it difficult to choose existing control mechanisms or to propose new ones.

Therefore, this paper focuses on the efficient execution of multiprogrammed workloads on a real instance of a clustered core-LLC organization. However, the

Table 1 Summary of papers on resource management with real machine experimentation

	Processor	Resources	Goal
Merlin [19]	Intel Westmere	LLC_I, IC, MC	Consolidation
Cook et al. [12]	Intel Sandy Bridge	LLC_I	Performance
Sun et al. [13]	Intel Sandy Bridge	LLC_I, $Pref_{hw}$	Performance
Pons et al. [25]	Intel Sandy Bridge	LLC_I	Turnaround time
Heracles [20]	Intel Haswell	LLC_I, Freq, Net	Consolidation
Ginseng [26]	Intel Haswell	LLC_I	Social welfare
Dirigent [22]	Intel Haswell	LLC_I, Freq	Consolidation
Selfa et al. [16]	Intel Haswell	LLC_I	Fairness
vCAT [24]	Intel Haswell	LLC_I	WCET
DCAPS [8], DCAT [14]	Intel Haswell	LLC_I	Performance
Chen et al. [23]	Intel Haswell	LLC_I, Freq, BW_{disk}, #Cores, Net	Consolidation
Kpart [7]	Intel Broadwell	LLC_I	Performance
Dicer [21]	Intel Broadwell	LLC_I	Consolidation
SWAP [4]	Cavium ThunderX	LLC	Performance
Kim et al. [15], CPPF [10]	Intel Skylake-SP	LLC_{NI}	Performance
Hypart [11]	Intel Skylake-SP	BW_{mem}	Performance
Copart [18]	Intel Skylake-SP	LLC_{NI}, BW_{mem}	Consolidation
EMBA [27]	Intel Skylake-SP	BW_{mem}	Consolidation
LFOC [17]	Intel Skylake-SP	LLC_{NI}	Fairness

LLC, last-level cache, which can be inclusive (I) or non-inclusive (NI); BW_{mem}, memory bandwidth; IC, interconnect; MC, memory controller; Freq, core frequency; BW_{disk}, disk bandwidth; #Cores, number of cores; $Pref_{hw}$, hardware prefetcher; Net, Internet connection. The rows are arranged in chronological order of the processor on which the mechanisms are applied

ideas used in the design could be applied to other processors, even with shared LLC. We propose very simple mechanisms, with negligible software overhead, that rely on existing monitoring and control support and require no hardware or operating system changes. The goal is to improve system performance and thread fairness by selecting certain cores, for which the space they occupy in the LLC and/or the memory bandwidth available to them will be dynamically limited or increased.

This paper makes the following contributions:

- We characterize the execution of the single-threaded, memory-intensive benchmarks of the SPEC CPU2006 and CPU2017 suites on an AMD Rome processor. The performance of each application running separately is studied in regard to both the allocated LLC capacity and memory bandwidth. Interference among applications in multiprogrammed workloads is also analyzed. Two key findings emerge. First, by limiting cache allocation we have found applications whose performance is barely affected, but whose memory bandwidth consumption increases significantly, negatively affecting system performance. Second, to estimate global memory bandwidth saturation it is best to look at memory access latency, because when off-chip memory traffic enters into a saturation regime, small increases in bandwidth can lead to significant increases in memory access latency.

Table 2 Main features of the selected server

Processor	AMD EPYC 7702P
Cores × Threads	64 × 2
L1 cache (I, D)	8-ways, 32 KiB (per core)
L2 cache	8-ways, 512 KiB (per core)
LLC	16-ways, 16 MiB, non-inclusive (per CCX)
Main memory	256 GiB DDR4, 8 channels, nominal peak BW: 204.8 GB/s
TurboBoost	Disabled
Hyperthreading	Disabled (1 thread/core), except for the core executing Balancer, see Sect. 4
OS	CentOS 8.2, kernel 4.18.0

- Based on the characterization findings, we propose Balancer, a new set of mechanisms that controls LLC space and memory traffic allocations to improve system performance and fairness when running multiprogrammed workloads.
- We evaluate Balancer with the aforementioned workload and processor and compare it with other state-of-the art LLC partitioning proposals. Balancer tuned for performance improves the average speedup of the applications by 7.1% and reduces the execution unfairness by 18.6%, with respect to the uncontrolled system. Balancer tuned for fairness reduces the execution unfairness by 64.5% in exchange for a 1.3% slowdown.

This paper is organized as follows. Section 2 describes the execution environment, the monitoring and control tools, and the workloads used in the characterization and the evaluation. Section 3 analyzes the impact of LLC occupancy and main memory traffic on the execution of multiprogrammed workloads. Section 4 details the Balancer mechanisms. Section 5 shows and analyzes the experimental results regarding performance and fairness. Section 6 compares our proposal with the related work, and Sect. 7 concludes. Appendix A provides information on how to obtain Balancer code and reproduce all the experiments.

2 Experimental environment and methodology

This section shows the main features of the selected server, the monitoring and control tools, and the employed workloads.

2.1 AMD Rome core organization

AMD launched in 2019 a family of processors aimed at the high-performance server segment. These models are based on the Zen 2 microarchitecture, code-named Rome [1]. Specifically, we have selected a server with a 64-core EPYC 7702P processor. The main characteristics of this system are specified in Table 2.

The 7702P processor is made up of up to eight compute core dies (CCDs) that are connected to each other and to off-chip memory via an I/O die (Fig. 1).

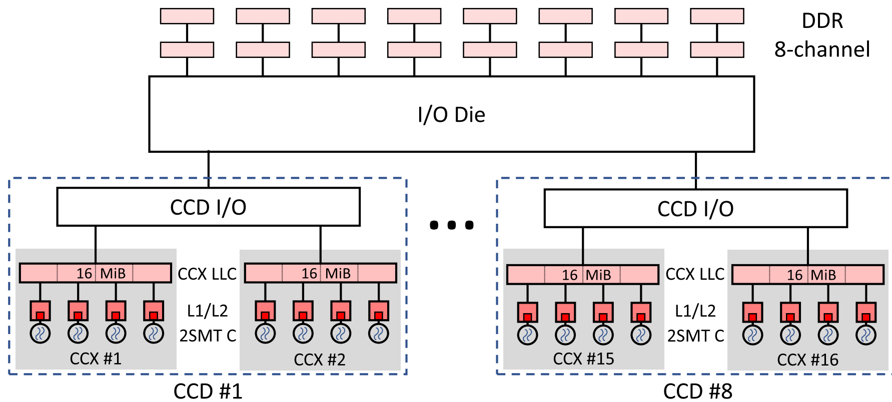

Fig. 1 AMD Rome 7702P clustered memory hierarchy. The multichip module has nine dies: eight CCD dies and one I/O die. In total there are 64 2-SMT cores (2SMT C) organized in 8 CCDs, each one with 2 CCX. Each CCX has a 16 MiB LLC shared by four cores [1] (color figure online)

Each CCD integrates two core complex (CCX) units that share an I/O connection. In turn, each CCX has four cores capable of executing eight threads sharing a 16 MiB victim LLC, i.e., the LLC gets populated with the cache blocks evicted from the four L2 caches of a CCX. This non-inclusive content management is also implemented in other recent processors such as ARM Neoverse [30] and Intel Skylake-SP [31]. Although there are 256 MiB of LLC in total, note that the four cores of a CCX cannot store cache blocks outside of their 16 MiB LLC.

2.2 Monitoring & control tools

In order to characterize the applications and then guide the Balancer control mechanisms, we use hardware counters that are read through their corresponding model-specific registers (MSR) [32]. GNU/Linux makes the MSR registers accessible through access to the files `/dev/cpu/[0-N-1]/msr`, where `N` is the number of cores. Hardware counters record events that occur during the execution of an application. For example, we can measure retired instructions, elapsed cycles, or misses experienced in the LLC of the CCX where an application is executed.

Moreover, there are specific banks of MSRs registers, belonging to *AMD64 Technology Platform Quality of Service Extensions* (AMD QoSE) [33] devoted to monitoring and enforcing limits on LLC allocation and memory *read* bandwidth on a per-thread basis. This is achieved with a 16-bit per-thread binary mask. Each bit of the mask enables each thread to use a particular sixteenth fraction of the LLC (1/16 = 1 MiB). Several threads can mark the same fraction(s), implying competitive sharing of the same LLC subset. Similarly, the memory *read* bandwidth can be limited per thread. This is achieved by writing an unsigned integer

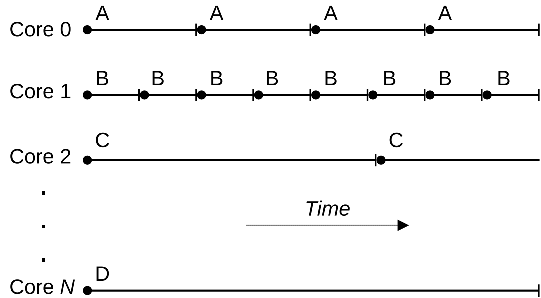

Fig. 2 Example of applications completions in a mix execution (color figure online)

to a specific MSR register, that sets a maximum read bandwidth in 1/8 GB/s increments.

2.3 Workload

We use multiprogrammed workloads [34, 35] composed of a subset of the benchmarks from the SPEC CPU2006 and CPU2017 suites.[1] Specifically, we selected the 33 memory-intensive single-threaded applications as suggested in [36]. These applications have been compiled with the GCC version and flags recommended by SPEC [37, 38]. We will first characterize each application running alone on the system and then with other independent applications in a variety of multiprogrammed situations. For the latter we generated one hundred *mixes*, which we use in the characterization and evaluation sections, made up of applications from the SPEC subset. In addition, we generated another ten different *mixes* that we use in Sect. 5.2 to adjust the parameters of our proposal.

Each mix consists of 64 instances (one for each core) randomly chosen from the 33 applications, so that any of the 33 applications have the same probability of occurrence, regardless of their execution time. For each application, we randomly selected a *reference* input data set from among those offered by SPEC (excluding *test* and *train* input data sets). Hence, zero, one, or more instances of the same application, with the same or different inputs, may appear in a particular mix.

During a mix execution, each application/input pair is pinned to a different core. A mix ends when its slowest application finishes its first execution (e.g., application D in Fig. 2). The rest of the applications are restarted with the same input and on the same core as they finish. Only *complete* executions are taken into account; this avoids over-representing the first phases of an application. In addition, in order to characterize each application, its behavior is first averaged within each mix, considering all the cores in which it appears and all the instances running on them; the same application in another mix with different partners may be executed a different number of times in a different number of cores, and might behave in a different way.

[1] Both SPEC CPU2006 and SPEC CPU2017 can be ordered at the following URL: https://www.spec.org/order.html.

Table 3 Metrics calculation from hardware counters [32, 33]

Metric	Formula	Scope
CPI	PMCx076 / PMCx0C0	Core
DMPKI	PMCx043 / (PMCx0C0 / 1000)	Core
MPKI	L3PMCx06 / (PMCx0C0 / 1000)	Core
HPKI	(PMCx043 + PMCx071 + PMCx05A) / (PMCx0C0 / 1000)	Core
L3Lat	(L3PMCx90 * 16) / L3PMCx9A	CCX
L3Occ	QOS L3 Occupancy	Core
rBW	((L3PMCx06 * 64) / 2^{30}) / Time	Core

Consequently, to obtain the metrics of an application we proceed as follows: first for each mix, compute the average of each application (all the completed instances in all cores); then for each application compute the average again over the values obtained across the one hundred mixes.

This method of mixing and collecting results has been used, except for the variation in inputs, in previous works [8, 10, 15].

3 Characterization

Regardless of whether the applications are run alone or in a multiprogrammed way, they are characterized by seven metrics: (1) CPI, cycles per instruction, (2) DMPKI, demand misses in the LLC per thousand instructions, (3) MPKI, total misses (demand + prefetch) in the LLC per thousand instructions, (4) HPKI, total hits (demand + prefetch) in the LLC per thousand instructions, (5) L3Lat, memory latency of LLC read misses, common to the four CCX cores, (6) L3Occ, average LLC occupancy, and (7) rBW, read traffic with memory, in GB/s. Table 3 shows the formulas used to calculate each metric from specific hardware counters of core or CCX scope.

3.1 Performance versus LLC capacity

In this subsection, we characterize the behavior of the applications *running alone* when varying the LLC capacity from 0 to 16 MiB with 1 MiB steps, with hardware prefetching enabled.

Figure 3 shows in the graphs, from left to right, CPI, DMPKI, and MPKI for each allocated LLC capacity. Each graph shows three lines corresponding to `410.bwaves` (blue), `471.omnetpp` (green), and `554.roms` (red), representative of the three main trends observed in all applications.

Most cache partitioning mechanisms proposed so far use miss and speedup curves, similar to those of CPI and DMPKI shown in Fig. 3, to decide how much LLC space should be allocated to each application [7, 17, 39]. In these two graphs we can distinguish two behaviors. On the one hand, `471.omnetpp` clearly takes

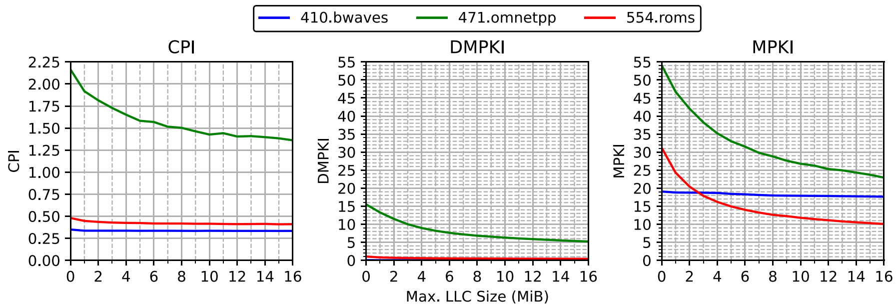

Fig. 3 CPI, DMPKI, and MPKI for increasing LLC allocation limits (1/16 MiB steps) (color figure online)

advantage of its available space in the LLC: CPI and DMPKI decrease significantly as the allocated space in the LLC increases. We can say that the behavior of `471.omnetpp` is sensitive to the available LLC size.

In contrast, `410.bwaves` and `554.roms` clearly waste the space they take in the LLC. Both metrics remain virtually constant as the LLC allocation bounds extend. We would say that the behavior of these two applications is insensitive to the available LLC size. Consequently, partitioning mechanisms based on these metrics would take cache space away from applications such as `410.bwaves` and `554.roms` and give more space to applications, such as `471.omnetpp`.

But if we analyze the MPKI graph, the miss rate considering both processor and prefetch requests, while `471.omnetpp` and `410.bwaves` maintain the same behavior, `554.roms` now shows a large decrease in MPKI as the available LLC space increases. Therefore, the MPKI metric would lead us to classify `554.roms` as sensitive to LLC size, contrary to the DMPKI and CPI metrics. This behavior is due to the prefetcher being effective in preloading the private caches with the data to be used, which eliminates demand misses. In other words, the prefetcher reuses the data stored in the LLC and, therefore, the more capacity the LLC has, the higher the hit rate. As a result, giving more LLC capacity to this type of applications does not imply a direct benefit for them but it does for the system, since it decreases the traffic with memory.

As far as we know, the behavior observed in `554.roms` with respect to the MPKI metric has not been previously highlighted and therefore it has not been considered when designing resource allocation mechanisms. Balancer will consider these applications as being cache sensitive and therefore will not limit their available LLC space so as not to increase bandwidth consumption. As we will show in Sect. 4.2, higher bandwidth consumption can increase memory access latency, which in turn implies a performance degradation of all applications running on the system.

Characterization highlights. *The MPKI metric, or its complementary HPKI, measures the benefit associated with LLC occupancy more comprehensively than CPI or DMPKI, which are the metrics commonly used in previous work. The*

Table 4 Activation of AMD EPYC 7702P components according to the number of Triad instances that are executed together with the application to be characterized. Recall that this processor integrates 64 cores organized in eight CCDs, each with two CCXs, which in turn have four cores each

# Triads	# active CCD	# active CCX per CCD	# active cores per CCX
0	1/8	1/2	1/4
3	4/8	1/2	1/4
7	8/8	1/2	1/4
15	8/8	2/2	1/4
31	8/8	2/2	2/4
63	8/8	2/2	4/4

variation in CPI or DMPKI only reflects the benefit that affects the application itself, while MPKI or HPKI, in addition, reflects the benefit that is achieved for the system.

3.2 Performance versus memory bandwidth

Next, we characterize the behavior of applications *not running alone* with increasing throttling of the available memory bandwidth or, in other words, as memory contention grows due to increasing aggregate traffic. For this purpose, we have used the Triad application.

Triad is a kernel of the STREAM[2] benchmark [40]. STREAM is considered as the de-facto benchmark to measure sustainable main memory bandwidth. It is a simple synthetic program that we use to generate data traffic between the CPU and main memory.

Triad performs simple operations on vectors: $A[j] = B[j] + scalar * C[j]$. Vectors A, B, and C are larger than L2 + LLC to ensure that there is no data reuse. We ran each of the selected applications alongside 0, 3, 7, 15, 31, and 63 Triad instances. The application to be characterized is pinned on the first core of the first CCX, and each Triad is pinned on another core, trying to occupy the maximum number of CCDs with an even split between CCXs. This thread scheduling runs the application on the AMD EPYC 7702P in configurations that first increase the number of active CCDs (from one to four and eight), then increase the number of active CCX per CCD (from one to two), and finally the number of cores per CCX (from one to two and four), see Table 4.

Figure 4 shows read traffic with main memory (gray bars, left Y-axis) and memory latency (black bars, right Y-axis), both averaged for all applications vs. the number of co-executed Triads (X-axis). The read traffic saturation point is reached at around 105 GB/s with only seven Triad instances. However, latency continues to grow as the number of Triad instances increases beyond seven, from 378 to 676 cycles with equal data traffic.

[2] The STREAM source code is available at the following URL: https://www.cs.virginia.edu/stream/FTP/Code/.

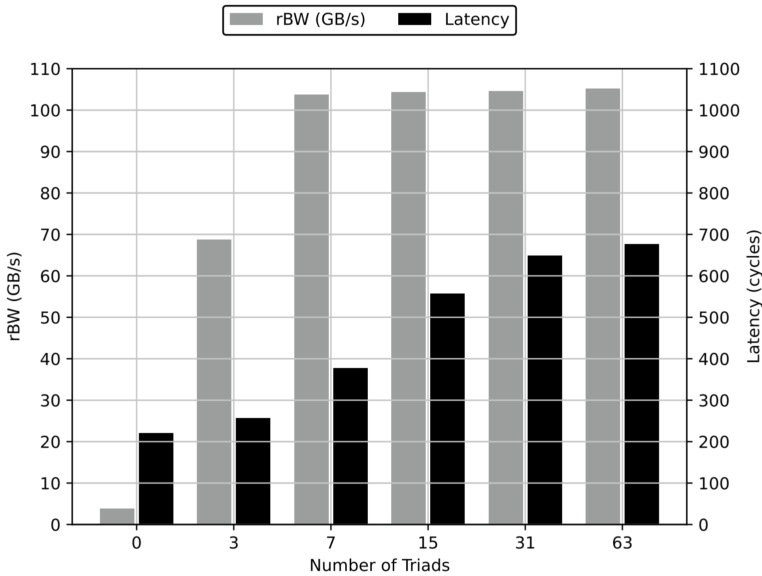

Fig. 4 Read memory traffic (left *Y*-axis) and memory access latency (right Y-axis) vs. number of Triads (X-axis) (color figure online)

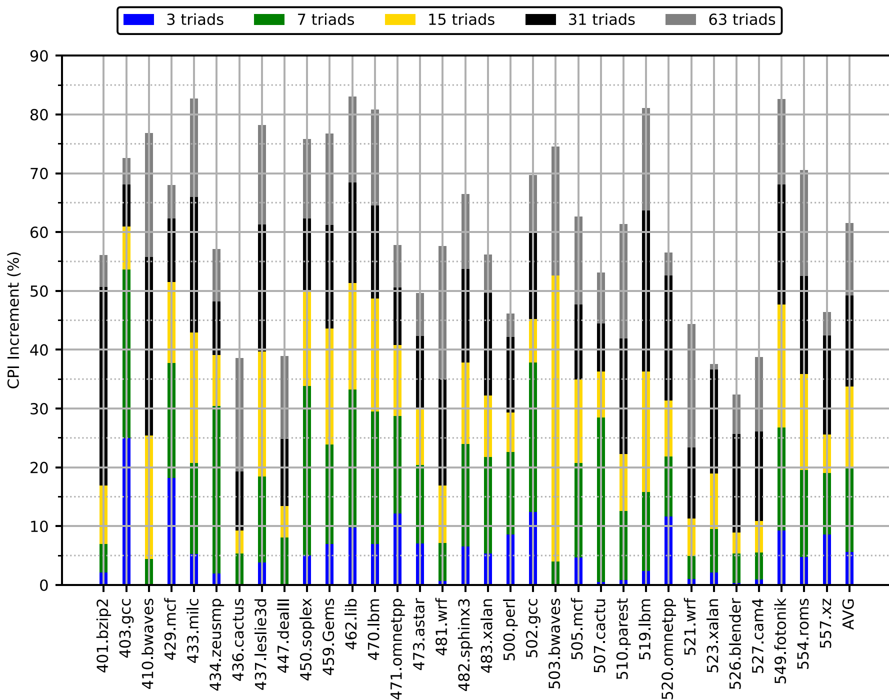

Fig. 5 CPI increase when running with multiple Triads (color figure online)

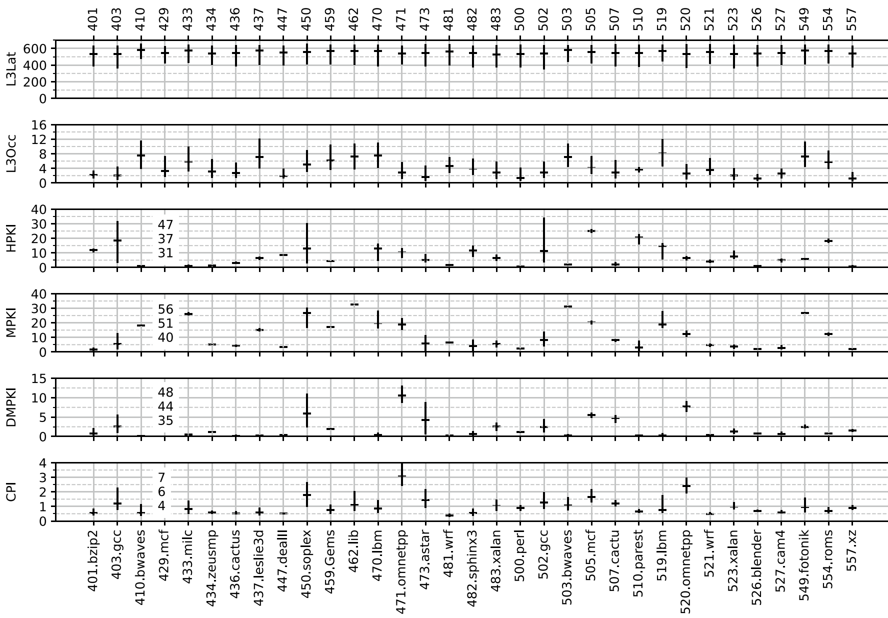

Fig. 6 CPI, DMPKI, MPKI, HPKI, L3Occ, and L3Lat. For each metric and application, the mean value and a vertical bar linking the minimum and maximum values are shown (color figure online)

Figure 5 shows the performance impact of increased latency. For each application it can be seen how its CPI increases as memory contention grows. On average, the progressive growth of traffic induces an increase in execution time of 5, 20, 34, 49, and 61% when an application contends with 3, 7, 15, 31, and 63 Triads, respectively. In 5 applications, increases of more than 80% are observed when co-executing with 63 Triads.

Characterization highlights. *Memory access latency is a better indicator than memory traffic for assessing memory contention. The increase in latency is a more direct measure of the impact on application execution time. In addition, memory latency increases if the request rate increases beyond the point of traffic saturation, allowing different degrees of contention to be identified.*

3.3 Multiprogrammed workload

Finally, we analyze the behavior of the applications on a fully loaded system, *running one application on each of the 64 cores*. Each application appears about two hundred times in the one hundred mixes, and is run under different conditions, determined by the other sixty-three co-executing applications.

Figure 6 shows the averages of the metrics (CPI, DMPKI, etc.) for those two hundred instances, along with a vertical bar linking the minimum and maximum values. Average memory latency is between 500 and 600 cycles for all applications,

reaching peaks of 650 for most of them. This indicates that memory traffic is always well above the saturation point.

Regarding LLC occupancy, application behavior is very diverse. Eight applications occupy on average more than 6 MiB, three of them reaching maximums of 12 MiB, while five other applications occupy on average less than 2 MiB. The variation among runs of the same application is also large. The difference between the minimum and maximum is greater than 3 MiB in 24 applications.

The benefit applications get from the space they occupy in the LLC is also very diverse as noted in Sect. 3.1. Applications such as `410.bwaves`, `433.milc`, `434.zeusmp`, `462.libquantum`, and `481.wrf` waste the space they occupy in the LLC since HPKI is practically zero in all their executions, regardless of the LLC capacity they fill. On the contrary, applications such as `471.omnetpp`, `403.gcc`, `429.mcf`, `450.soplex`, and `473.astar` show significant differences between maximum and minimum values in occupancy, CPI, MPKI, and HPKI. These applications take advantage of the space in LLC in a clear way, reducing their execution time if they get more space. Finally, in applications such as `436.cactusAMD`, `437.leslie3d`, `519.lbm`, `549.fotonik3d`, and `554.roms` we note significant variations in LLC occupancy that do not translate into CPI and DMPKI differences, but do translate into MPKI and HPKI differences. Therefore, these applications do not get a direct benefit by occupying more space in the LLC, but they can bring a benefit to the system by reducing the traffic with the main memory.

Characterization highlights. *When running multiprogrammed workloads, it is common for the traffic generated by memory requests to congest the DRAM channels. This results in high memory latencies, which in turn affects application execution time. The LLC space occupied by applications is very diverse and varies a lot among executions. The behavior reported in Sect. 3.1 is repeated when running applications on multiprogrammed workloads.*

4 Balancer: bandwidth allocation and cache partitioning

This section introduces Balancer, a set of new mechanisms for allocating shared resources to the cores of a multicore processor. The first one, CCO (Control of LLC Occupancy), manages the sharing of space in the LLC. The second, CMT (Control of Memory Traffic), manages the amount of read memory bandwidth. Both can be tuned to act together in LLC occupancy and read bandwidth (CCO+CMT).

Unlike other proposals, Balancer can be easily applied to clustered organizations because it can make different decisions in each cluster in a decentralized manner, in response to their particular cache utilization and bandwidth consumption.

These mechanisms have been implemented on a server based on an AMD 7702P processor. The scripts have been developed in Python3, and are executed in user space (Appendix A). A specific thread, called Balancer, executes such scripts every second. We define epoch as the time interval that elapses between two executions of the scripts triggered by Balancer. First, the monitored events recorded in the hardware counters are read, and the metrics of interest are calculated. The last ten values of these metrics are then averaged. This calculation prevents one-off peaks

from triggering the imposition of constraints, and also prevents events in the distant past that are no longer relevant from affecting the values of the metrics. The Balancer thread is pinned to the first core of the first CCX, being this core the only one that has hyperthreading enabled. The other cores run their applications with hyperthreading disabled and zero overhead due to the execution of Balancer. This spatial decision to assign the Balancer thread to one or another CCX is irrelevant since its overhead is absolutely negligible. We have measured with perf [41] the CPU time consumed by the Balancer thread (monitoring + control). This time represents 0.1% of the execution time of all system cores.

4.1 Control of LLC occupancy (CCO)

4.1.1 Motivation

LLC space can become a scarce resource if there is competition among applications. Considering the overall benefit to the system, it would be desirable to allocate more space to the applications that can take the most advantage of it. The idea is to foster a good reuse of the data stored in the LLC to improve the performance of individual cores and, in the process, generate less main memory traffic. We calculate the reuse ratio as the number of hits that each block fetched to LLC receives, i.e., $\text{Hits}_{\text{LLC}}/\text{Misses}_{\text{LLC}}$. However, different applications may need more or less LLC space to achieve the same reuse ratio. It would be desirable to give priority to those applications that achieve a higher reuse ratio while occupying less LLC space.

4.1.2 Proposal

We define a new metric, the *number of hits per miss and MiB occupied* (HpMO), to compare the LLC space usage efficiency of applications. This metric is directly proportional to the reuse ratio and inversely proportional to the space occupied in LLC. Hit, miss, and occupancy rates (HPKI, MPKI, and L3Occ) are sampled in each CCX LLC. For each core of a CCX, HpMO is calculated as follows:

$$\text{HpMO} = \frac{\text{Hits}_{\text{LLC}}}{\text{Misses}_{\text{LLC}} \cdot \text{Occupancy}_{\text{LLC}}} \tag{1}$$

HpMO quantifies for each core the profit obtained with the cache blocks it has in the LLC, either evicted from L2 or brought by prefetching. The higher the HpMO, the better the space utilization in the LLC and vice versa. For example, a core with a 20% reuse ratio and 5 MiB occupancy has a very low HpMO value of 0.05, evidencing little benefit from LLC occupied space.

Balancer considers that a core is using the space it occupies in LLC inefficiently if its HpMO is less than a certain threshold *X*. In this case, the core will be *constrained* in a shared 1 MiB partition (minimum LLC fraction, as described in Sect. 2.2). This LLC partition is shared by all cores, constrained or not. Consequently, the LLC size occupied by all restricted cores is at most 1 MiB, leaving the other 15 MiB for the exclusive use of the unrestricted cores. In this way, CCO allows other applications to

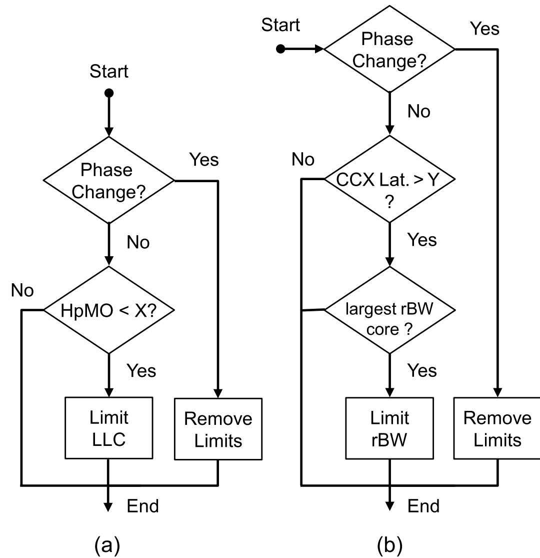

Fig. 7 COC **a** and CMT **b** control algorithms (color figure online)

take advantage of the space left over in the LLC, reducing MPKI and main memory traffic. Figure 7a shows the CCO control algorithm for one core, which applies to all cores in the system every epoch. The proper values for the HpMO threshold depend slightly on the mechanism target, performance or fairness, and will be studied in Sect. 5.2.

In the unlikely event that there is only one unconstrained core in a CCX, no new limits will be imposed. Finally, constraints on a core are removed when a phase change is detected. We assume that a phase change has occurred when the behavior of the program with respect to the LLC has experienced a significant change, which justifies Balancer to re-evaluate the metrics of interest. Therefore, the metric to detect a phase change must be related to LLC, since a change in other metrics such as branch predictor misses or L1D accesses might not correlate with a significant change with respect to LLC. We used total requests (demand and prefetch) instead of misses because they are not affected by external behaviors, e.g., change in LLC partition or phase change in other applications. Balancer considers that a phase change exists if the number of LLC prefetch and demand requests per kilo instructions of a core (HPKI + MPKI) differs by more than 20% from the previous measurement.

4.2 Control of memory traffic (CMT)

4.2.1 Motivation

As we have seen in Sect. 3, memory latency is a good proxy of communication contention between the executing cores and main memory. That is, when memory traffic reaches the limit supported by the system, the increased rate of requests to main

memory translates into an increase in memory latency which, in turn, affects the performance of running applications, as shown in Fig. 4.

4.2.2 Proposal

Balancer considers that memory access congestion exists if the memory access latency exceeds a certain threshold *Y*. Figure 7b shows the CMT control algorithm for one core, which applies to all cores in the system every epoch. The average latency of the off-chip memory requests made by each CCX is monitored. If it exceeds the threshold *Y*, the read bandwidth consumption of each core in that CCX is examined. The core responsible for the largest one is selected, and its bandwidth limited to 2.5 GB/s. If the selected core was already limited, its bandwidth limit is further decreased by 10%, until reaching a minimum that would correspond to an equal distribution between cores, i.e., 1/64 of 105 GB/s, the peak rBW measured. Again, the proper values for the CCX latency threshold depend slightly on the mechanism target, performance or fairness, and will be studied in Sect. 5.2.

As in CCO, constraints on a core are removed when a phase change is detected.

4.3 Balancer: simultaneous control of LLC occupancy and memory traffic (CCO+CMT)

CCO and CMT can be combined to obtain better performance than that achieved by each mechanism separately. CCO is conservative in that it limits occupancy to cores that waste LLC, in the hope that the freed capacity will be leveraged by the remaining cores to improve overall performance. In contrast, CMT is aggressive in that it limits traffic in applications that consume high memory bandwidth, even though it may be contributing to good performance. Therefore, we propose combining both controls, but first applying CCO, to gradually improve LLC occupancy in successive epochs and then CMT. Thus, when a control epoch starts, firstly CCO acts: the HpMO of each core is compared with threshold *X* in each CCX, and the cores without LLC occupancy limits that are below the threshold are confined in the 1 MiB partition. Secondly, CMT will act as explained only on those CCXs that have not experienced new confinements, i.e., on CCXs with latency above threshold *Y* the highest traffic core is selected and its bandwidth limited.

5 Balancer evaluation

In this section, we evaluate our proposal (Balancer), and compare it with a system without control (Uncontrolled) and with three control mechanisms using cache partitioning: (1) equal sharing of resources through static allocation, i.e., 4 MiB of LLC and 1.6 GB/s bandwidth per core (Static), (2) static UCP guided by DMPKI (UCPd), and (3) static UCP guided by MPKI (UCPm). UCP stands for Utility-based Cache Partitioning [39].

5.1 Metrics and baseline system

As a baseline system we will use Static, which is positioned at one extreme of the performance/fairness tradeoff: it does not allow dynamic sharing (i.e., surplus hardware resources cannot be exploited by cores with scarcity), and it is intrinsically fair in terms of hardware resources (i.e., equal resource partitioning). Of course, reporting values relative to the baseline system does not exclude mutual comparison between the rest of the mechanisms, but it facilitates discussion when focusing on individual application behaviors, which in some metrics have very different absolute values.

As a performance metric we use the speedup relative to the baseline system.

To assess execution fairness, we use the metric M_1 defined by Kim et al. [9]. Unlike other metrics such as the harmonic CPI, M_1 is a pure unfairness metric, independent of performance rewards. For a mechanism *mec* controlling the execution of a mix of applications, M_1 is calculated as:

$$M_1(\text{mec}) = \sum_i \sum_j \left(\frac{\text{IPC_sta}_i}{\text{IPC_mec}_i} - \frac{\text{IPC_sta}_j}{\text{IPC_mec}_j} \right) \quad (2)$$

where IPC_mec_i is the IPC of application i when running in a system controlled by the mec mechanism, and IPC_sta_i is the IPC when running in the baseline system.

5.2 Design space exploration

For decision-making, Balancer uses two thresholds on HpMO and CCX latency values, Sects. 4.1 and 4.2, respectively. To analyze their impact on performance and fairness, we ran ten new mixes under Balancer control using a set of thresholds for HpMO (from 0.03 to 0.07 with steps of 0.01) and CCX latency (from 300 to 450 with steps of 50). Figure 8 shows the average value of speedup (*Y*-axis) and M1 (*X*-axis) obtained by each configuration across all the mixes in our workload. The results of each evaluated configuration are shown with the combination of a shape and a color. The shapes specify the latency thresholds while the colors indicate the HpMO thresholds. For example, a yellow square represents the speedup and unfairness for a configuration with thresholds of 0.05 and 400 cycles for the HpMO and latency thresholds, respectively. The gray shapes show the results of CTM-only Balancer while the circles show the results of CCO-only Balancer. The horizontal and vertical dashed lines represent the speedup and unfairness results, respectively, for Uncontrolled.

Variations in the CCX memory latency threshold significantly affect fairness and performance. Reductions in the latency threshold improve fairness, as the CMT mechanism imposes more traffic constraints on cores, approaching an equal sharing of memory bandwidth among all cores. CMT always achieves drastic reductions in unfairness, although in some configurations it produces

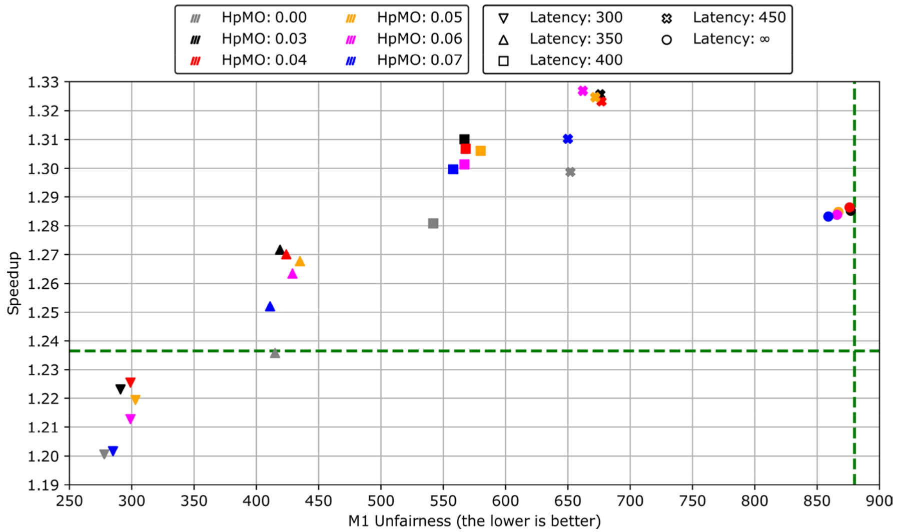

Fig. 8 Speedup and unfairness for balancer with different thresholds. Different colors and shapes represent results for different HpMO and latency thresholds, respectively. Green dashed lines correspond to an Uncontrolled system (color figure online)

Table 5 Balancer-P and Balancer-F thresholds (color figure online)

Metrics	Balancer-P	Balancer-F
HpMO, hits/(misses·MiB)	$X = 0.06$	$X = 0.04$
CCX mem. latency, cycles	$Y = 450$	$Y = 300$

performance losses. In the best case, CMT manages to divide the M1 metric by 3.17 with respect to Uncontrolled.

On the other hand, variations in the HpMO threshold affect performance but have negligible effects on fairness. CCO always has a positive impact on performance. It achieves a speedup varying between 0.1 and 4.0% with a small loss of fairness.

By combining CCO and CMT and properly selecting thresholds, the Balancer resource control can be directed to different targets, matching the performance/fairness tradeoff at will. For their detailed evaluation with the one hundred mixes cited in Sect. 2.3 we select two control solutions, Balancer-P and Balancer-F (Table 5). *Balancer-P* is intended to optimize performance, uses values of 0.06 and 450 for the HpMO and latency thresholds respectively, and achieves a speedup of 7.01% with respect to Uncontrolled, with a 32.9% reduction in unfairness. *Balancer-F* is designed to optimize fairness, uses values of 0.04 and 300 for the HpMO and latency thresholds respectively, and achieves a 66.0% reduction in unfairness, although it produces a 1.2% performance loss with respect to Uncontrolled. As an example of minimum unfairness we have discarded the CMT-only system (gray inverted triangle in Fig. 8), because it achieves a slight reduction in unfairness but with a significant loss of performance.

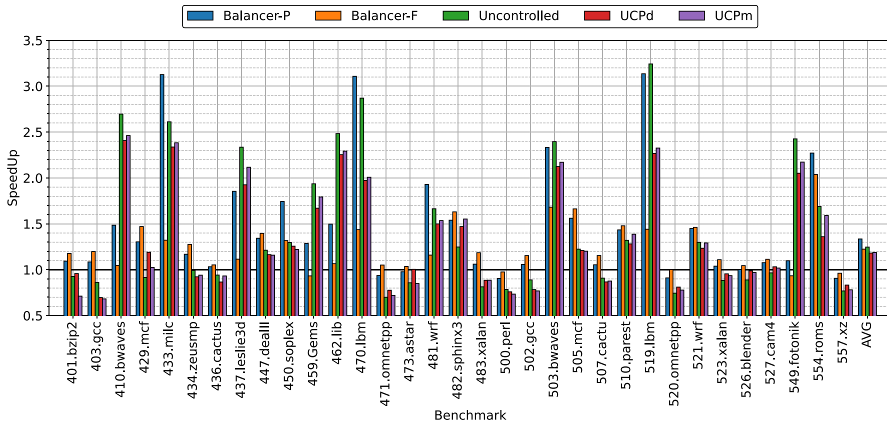

Fig. 9 Speedup of the selected SPEC CPU2006 and CPU2017 applications for all control mechanisms relative to Static (color figure online)

5.3 Performance

Figure 9 shows, for each application, the speedup obtained by all mechanisms with respect to the Static baseline system. The last group of bars on the right shows that, on average, all mechanisms improve performance against Static. However, only Balancer-P outperforms Uncontrolled. Balancer-P focuses on improving overall system performance. Since it limits bandwidth to those cores that cause high memory latencies, it causes 7 applications out of 33 to lose performance with respect to Uncontrolled, being `549.fotonik3d_r` the worst case with a 0.45 slowdown. However, Balancer-P outperforms Uncontrolled in 14 out of 18 SPEC CPU2006 applications and 12 out of 15 SPEC CPU2017 applications, and achieves an average improvement of 7.1%, with a maximum speedup of 42.4% on `429.mcf`.

Balancer-F, on the other hand, improves fairness by more aggressively limiting main memory traffic, resulting in performance losses relative to Uncontrolled in 10 applications out of 33 selected. Yet Balancer-P outperforms Uncontrolled in 11 out of 18 SPEC CPU2006 applications and in 12 out of 15 SPEC CPU2017 applications, with average performance only 1.3% worse.

Notice that Balancer-P slows down five applications with respect to Static (`471.omnetpp`, `473.astar`, `500.perl`, `520.omnetpp`, `557.xz`). These are *weak* applications in the sense that they take up few resources when in competition with other applications, as can be seen from the L3Occ data in Fig. 6. Therefore, these applications achieve better performance if they are allocated 4 MiB of LLC without competition. The two Balancer configurations lose the least with respect to Static, with quite a difference in some cases with respect to all other mechanisms. In other words, Balancer manages to protect these *weak* applications better than the other mechanisms.

Table 6 Average M_1 execution fairness

Balancer-P	Balancer-F	Uncontrolled	UCPd	UCPm
693	298	851	634	698

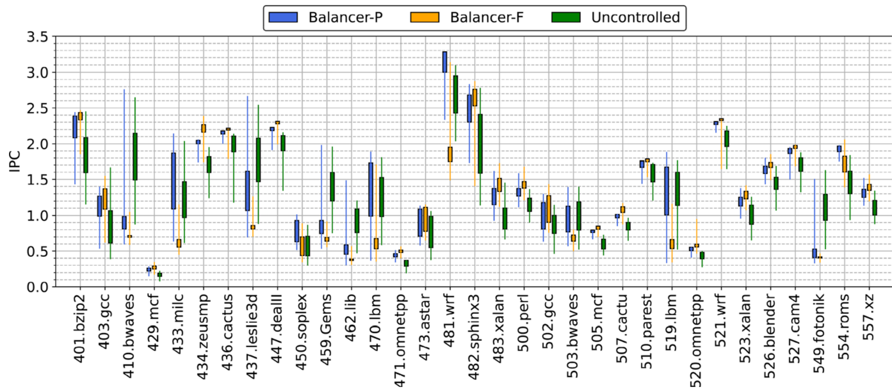

Fig. 10 IPC variability: 75th and 25th percentiles and maximum and minimum values (color figure online)

5.4 Fairness

Table 6 shows the average value of M_1 obtained by each mechanism for all the mixes in our workload. Note that M_1 is an unfairness metric and therefore the lower the better.

Uncontrolled presents the highest value of unfairness ($M_1 = 851$). This is an expected result as each application uses the resources it needs without regard to the impact on other applications. UCPd and UCPm manage to reduce the unfairness by 25% and 18% with respect to Uncontrolled, but at the cost of a significant loss of performance, as we have seen in the previous section. Balancer-F, on the other hand, achieves a much larger reduction in unfairness, 64.5% less than Uncontrolled, with a performance loss of only 1.3%. Even the performance optimized version, Balancer-P, manages to reduce the unfairness to values similar to those of UCPd and UCPm with a very significant performance improvement.

An important benefit of improving system fairness is that execution time predictability increases. Less variability in the execution time of applications facilitates scheduling decisions and minimizes unexpected charges for services.

Figure 10 shows, for each application, the maximum, minimum, 75th, and 25th percentiles of the IPC values obtained for all application instances run in the different workload mixes. Values are shown for the two Balancer configurations and for Uncontrolled.

Balancer-F reduces the variability in IPC with respect to Uncontrolled in all applications except `502.gcc`. The reduction is very significant in many applications. As an example, with Balance-F, the difference between the 25th and 75th

percentiles is less than 10.0% in 20 of the 33 applications, and greater than 20.0% in only 5 applications. With Uncontrolled, the difference is greater than 10.0% for all applications and greater than 20.0% for 23 out of 33. Balancer-P also reduces the variability compared to Uncontrolled in most applications, 26 out of 33, but to a lesser extent than Balancer-F.

5.5 Number of cores and scalability

An alternative to limiting memory bandwidth is to decrease the number of cores used to run applications. By loading the system with a smaller number of applications, each application has a larger fraction of resources at its disposal.

To test the impact on system performance of leaving cores idle, we ran 10,000 SPEC CPU2006 and CPU2017 applications on our system using 64 cores (all active, one pinned application per core), 56 cores (1 idle core per CCD) and 48 cores (1 idle core per CCX). In all experiments, the Balancer thread is active, and when an application terminates and frees a core, the next application to run on it is the one with the lowest cumulative execution time. This ensures that all applications are represented uniformly, regardless of their individual running time.

Running the 10,000 applications on 64 cores took 15 h 52 min, on 56 cores (87% of our processor's total capacity) it took 17 h 27 min (10% more), and on 48 cores (75% of our processor's total cores) it took 18 h 10 min (13% more). Therefore, we have not seen any performance improvement from leaving cores idle. In terms of performance it is best to keep the system at its maximum possible load.

Works such as García et al. [17] or Xiao et al. [10] use a maximum of 80% and 40% of the processor cores respectively to evaluate their mechanisms. García et al. uses 8, 12 and 16 cores to analyze the impact of the number of applications on fairness. Xiao et al. uses only 8 cores because they do not have enough masks to run their algorithm with a larger number of cores. Our algorithm does not have any limitation on masks. Nor have we perceived a performance improvement that would justify the use of fewer cores than the maximum available in our processor.

6 Related work

Most previous proposals for shared resource control act on the LLC to improve overall system performance or fairness [4, 7–18, 39], or turnaround time [25]. Among them we highlight KPART [7], DCAPS [8] and LFOC [17]. They use demand miss rate and IPC curves, which are dynamically computed, to guide application clustering and LLC partitioning. LFOC identifies in a first step the applications with IPC insensitive to the LLC capacity and isolates them in a single partition with two cache ways. Balancer uses this same technique with two differences: i) LFOC uses DMPKI to guide this decision while Balancer uses HpMO. This implies that in some cases LFOC may increase bandwidth consumption. ii) Balancer uses a single

cache way partition to isolate applications because its selection algorithm is more restrictive than that of LFOC.

Some of these proposals also use memory traffic to drive their decisions [7, 19–21, 23]. However, only CoPart acts directly on this resource in addition to acting on LLC [18]. Our proposal acts on both resources. So, CoPart is the proposal that most closely resembles ours. CoPart acts on the LLC and memory traffic of a 16-core Intel Xeon to improve fairness. It first performs profiling to determine the sensitivity of each application to the slowdown it experiences when the cache or available bandwidth gets reduced. Then, applications are dynamically ranked according to whether they need more resources or are able to give up the ones they have. CoPart formulates LLC and memory bandwidth allocation as an economy problem, where resources are reallocated from one application to another using a heuristic looking for maximum fairness.

The main problem of all these mechanisms is the complexity of the control/monitoring actions that classify the applications. An example is the use of the miss rate curve. Usually, the mechanisms require the execution of each application with different resource limits to obtain such behavior models. This classification has to be repeated periodically to detect application phase changes. Moreover, this problem is more critical as the number of cores increases. In contrast, our proposals monitor simple metrics to detect inefficient use of LLC or excessive latency due to bandwidth abuse.

Another important limitation of these mechanisms is that their heuristics, when changing the allocation of resources to an application, only consider the achieved self-profit, without taking into account the impact on the system. On the contrary, the mechanisms proposed in Balancer identify inappropriate uses of resources in order to prevent them and improve system performance.

On the other hand, other proposals only act on memory traffic [11, 27, 42–46]. EMBA limits the memory traffic of the applications that use more bandwidth [27]. It does this progressively and observes the impact on system performance. The mechanism stops increasing the limitations when it detects a loss in performance. In addition, it restarts when it detects a phase change in one of the applications. The mechanism is tested with only eight cores, and unlike our proposal, it does not control LLC allocation.

PABST focuses on controlling memory traffic by restricting request rates and changing the priority of memory requests [42]. However, it is evaluated on a simulator, and requires extra hardware. In contrast, our proposal runs on real hardware. Other works that also use simulation instead of real hardware are [43–46]. These works propose new schedulers for the main memory controller to improve system performance and/or fairness.

Finally, we have considered comparison with several of these state-of-the-art LLC control mechanisms, but it has not been possible for several reasons. Most authors rely on closed source codes, making it very difficult to ensure that a third-party implementation is truthfully conforming to their approach. We are aware of only two works that provide open source codes, ElSayed et al. [7] and Pons et al. [25]. However, both mechanisms are intended for Intel processors with an inclusive LLC and no clustering of cores and resources. In contrast, the AMD organization uses

non-inclusive LLCs grouped in CCXs serving clusters of cores, which compete for main memory bandwidth through two levels of routing, first the CCD I/O and then the die I/O. Therefore, adjusting these mechanisms to a hierarchical organization requires in-depth changes that go beyond code adaptation. For example, it is necessary to decide whether to establish a single control mechanism throughout the system or a mechanism per shared LLC in every CCX. Moreover, it is necessary to find the AMD hardware counters, if they exist, equivalent to the Intel ones. In our attempt to port these mechanisms, we were unable to compile the code of Pons et al. and obtained unsatisfactory results when running that of ElSayed et al. For these reasons, we decided to leave them out of comparison.

7 Conclusions

This paper presents a detailed characterization of the execution of the subset of single-threaded, memory-intensive test programs of the SPEC CPU2006 and CPU2017 suites on an AMD Rome processor. This analysis focuses on the impact of available LLC and memory bandwidth on the performance of an application. We have identified a type of application whose performance is barely affected when its allocated LLC space decreases, but whose memory bandwidth consumption increases significantly, negatively affecting system performance. We have also found that memory access latency is a better indicator than memory traffic for assessing memory access contention. As far as we know, this is the first time that these findings have been highlighted.

From the characterization work, we have proposed strategies that impose limits on LLC space utilization and memory traffic to specific applications. These constraints improve performance and/or fairness of multiprogrammed workloads, on average, with respect to a system with no control. Specifically, Balancer-P, tuned for performance, improves IPC 7.1% and reduces unfairness 18.6% compared to the system without control, while Balancer-F, tuned for fairness, reduces unfairness 64.5% in exchange for a 1.3% loss in performance. Balancer requires no hardware or operating system modifications. Our proposal is the only one, to our knowledge, that controls LLC occupancy and memory traffic on an AMD processor with 64 cores organized in clusters.

Appendix A: Paper artifact description appendix

Artifact description

The paper reports experiments that characterize the execution of the single-threaded, memory-intensive benchmarks of the SPEC CPU2006 and CPU2017 suites on an AMD Rome processor. It also reports experiments to evaluate Balancer, a new set of mechanisms that controls LLC space and memory traffic allocations to improve system performance and fairness when running multiprogrammed workloads. We have

published in a public repository the Balancer code and the information necessary to reproduce the experiments for the characterization: https://github.com/agusnt/BALANCER.

Experimental framework

Hardware

We have used a server with a 64-core EPYC 7702P processor and 256 GiB of DDR4 DRAM. Both TurboBoost and Hyperthreading options were disabled. See Table 2 for more details.

Operating system

CentOS 8.2, kernel 4.18.0.

Compilers

Benchmarks have been compiled with gcc version 7.5.0.

Applications

The scripts that run the experiments are developed in Bash and Python 3.6.8.

Libraries and versions

The following Python libraries have been used: numpy, psutil, and json.

Input datasets

The workload chosen involves a subset of the single-threaded, memory-intensive applications from SPEC CPU2006 and CPU2017 suites and the Triad application of the STREAM benchmark. Both SPEC CPU2006 and SPEC CPU2017 can be ordered at the following URL: https://www.spec.org/order.html. The STREAM source code is available at the following URL: https://www.cs.virginia.edu/stream/FTP/Code/.

Authors' contribution ANT has performed experimentation and data processing. All authors have proposed experiments, analyzed their results, wrote the main manuscript text and reviewed the manuscript. ANT prepared the figures. PI and VVY sought funding for the project.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. All authors acknowledge support from grants (1) PID2019-105660RB-C21 / AEI / 10.13039/501100011033 from Agencia Estatal de Investigación (AEI) and European Regional Development Fund (ERDF), (2)

gaZ: T58_20R research group from Aragón Government and European Social Fund (ESF), and (3) 2014-2020 "Construyendo Europa desde Aragón" from European Regional Development Fund (ERDF).

Declarations

Conflict of interest The authors declare no competing interests.

Ethical approval Not applicable.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Mattioli M (2021) Rome to Milan, AMD continues its tour of Italy. IEEE Micro 41(4):78–83. https://doi.org/10.1109/MM.2021.3086541
2. Starke WJ, Thompto BW, Stuecheli JA, Moreira JE (2021) Ibm's power10 processor. IEEE Micro 41(2):7–14. https://doi.org/10.1109/MM.2021.3058632
3. Herdrich A, et al. (2016) Cache QoS: from concept to reality in the Intel® Xeon® processor E5-2600 v3 product family. In: HPCA, pp. 657–668
4. Wang X, et al. (2017) SWAP: effective fine-grain management of shared last-level caches with minimum hardware support. In: HPCA, pp. 121–132
5. Kashyap A (2020) High performance computing: tuning guide for AMD EPYC™ 7002 Series Processors. Pub. 56827, rev 1.0
6. Karamatas C (2022) AMD EPYC 7003 series microarchitecture overview. Pub. 57075, rev 3.0
7. El-Sayed N, et al. (2018) KPart: a hybrid cache partitioning-sharing technique for commodity multicores. In: HPCA, pp. 104–117
8. Xiang Y, et al. (2018) DCAPS: dynamic cache allocation with partial sharing. In: EuroSys
9. Kim S, et al. (2004) Fair cache sharing and partitioning in a chip multiprocessor architecture. In: PACT, pp. 111–122
10. Xiao J, et al. (2019) CPpf: a prefetch aware llc partitioning approach. In: ICPP
11. Park J, et al. (2018) Hypart: a hybrid technique for practical memory bandwidth partitioning on commodity servers. In: PACT
12. Cook H, et al. (2013) A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In: ISCA, pp. 308–319
13. Sun G, et al. (2019) Combining prefetch control and cache partitioning to improve multicore performance. In: IPDPS, pp. 953–962
14. Xu M, et al. (2018) DCat: dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In: EuroSys, pp. 1–13
15. Kim Y, et al. (2019) Application performance prediction and optimization under cache allocation technology. In: DATE, pp. 1285–1288
16. Selfa V, et al. (2017) Application clustering policies to address system fairness with Intel's cache allocation technology. In: PACT, pp. 194–205
17. Garcia-Garcia A, Saez JC, et al. (2019) LFOC: a lightweight fairness-oriented cache clustering policy for commodity multicores. In: ICPP
18. Park J, et al. (2019) CoPart: coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In: EuroSys
19. Tembey P, et al. (2014) Merlin: application- and platform-aware resource allocation in consolidated server systems. In: SOCC, pp. 1–14

20. Lo D et al (2016) Improving resource efficiency at scale with Heracles. ACM Trans Comput Syst 34(2):1
21. Nikas K, et al. (2019) DICER: diligent cache partitioning for efficient workload consolidation. In: ICPP. ICPP 2019
22. Zhu H, et al. (2016) Dirigent: enforcing qos for latency-critical tasks on shared multicore systems. In: ASPLOS, pp. 33–47
23. Chen S, et al. (2019) Parties: QoS-aware resource partitioning for multiple interactive services. In: Proc of the Twenty-Fourth Intl Conf on Architectural Support for Programming Languages and Operating Systems, pp. 107–120
24. Xu M, et al. (2017) vCAT: dynamic cache management using cat virtualization. In: RTAS, pp. 211–222
25. Pons L et al (2020) Phase-aware cache partitioning to target both turnaround time and system performance. IEEE Trans Parallel Distrib Syst 31(11):2556–2568
26. Funaro L, et al. (2016) Ginseng: market-driven llc allocation. In: USENIX ATC, pp. 295–308
27. Xiang Y, et al. (2019) Emba: efficient memory bandwidth allocation to improve performance on Intel commodity processor. In: ICPP
28. Fujitsu: A64FX® Microarchitecture Manual Ver. 1.6 (2021)
29. Naffziger S, et al. (2021) Pioneering chiplet technology and design for the AMD EPYC™ and Ryzen™ processor families : Industrial product. In: ISCA, pp. 57–70
30. Pellegrini A et al (2020) The arm neoverse N1 platform: building blocks for the next-gen cloud-to-edge infrastructure soc. IEEE Micro 40(2):53–62. https://doi.org/10.1109/MM.2020.2972222
31. Doweck J et al (2017) Inside 6th-generation Intel Core: new microarchitecture code-named Skylake. IEEE Micro 37(2):52–62. https://doi.org/10.1109/MM.2017.38
32. Advanced micro devices: processor programing reference (PRR) for AMD Family 17h Model 20h, Revision A1 Processors. Rev 3.07 (2020)
33. Advanced micro devices: AMD64 technology platform quality of service extensions. Pub. 56375, rev 1.01 (2018)
34. Agarwal A, Hennessy J, Horowitz M (1988) Cache performance of operating system and multiprogramming workloads. ACM Trans Comput Syst 6(4):393–431. https://doi.org/10.1145/48012.48037
35. Eyerman S, Eeckhout L (2008) System-level performance metrics for multiprogram workloads. IEEE Micro 28(3):42–53
36. Navarro-Torres A, et al. (2019) Memory hierarchy characterization of SPEC CPU2006 and SPEC CPU2017 on the Intel Xeon Skylake-SP. PLOS ONE, 1–24
37. Standard performance evaluation corporation: SPEC CPU 2006. https://www.spec.org/cpu2006/
38. Standard performance evaluation corporation: SPEC CPU 2017. https://www.spec.org/cpu2017/
39. Qureshi MK, et al. (2006) Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches. In: MICRO, pp. 423–432
40. McCalpin JD (1995) Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, 19–25
41. De Melo AC (2010) The new linux "perf" tools. In: Linux Kongress, Vol. 18. http://vger.kernel.org/~acme/perf/lk2010-perf-paper.pdf
42. Hower DR, et al. (2017) PABST: proportionally allocated bandwidth at the source and Target. In: HPCA, pp. 505–516
43. Ebrahimi E, et al. (2010) Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In: ASPLOS, pp. 335–346
44. Ipek E, et al. (2008) Self-optimizing memory controllers: a reinforcement learning approach. In: ISCA, pp. 39–50
45. Mutlu O, et al. (2007) Stall-time fair memory access scheduling for chip multiprocessors. MICRO, 146–160
46. Nesbit KJ, et al. (2006) Fair queuing memory systems. In: MICRO, pp. 208–222

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.