Co-scheduling HPC workloads on cache-partitioned CMP platforms

Guillaume Aupy, Anne Benoit, Brice Goglin, Loïc Pottier, Yves Robert

Co-scheduling HPC workloads on cache-partitioned CMP platforms

Guillaume Aupy[1], Anne Benoit[2,3], Brice Goglin[1], Loïc Pottier[2], and Yves Robert[2,4]

[1]Inria, Université de Bordeaux, France
[2]Laboratoire LIP, École Normale Supérieure de Lyon, France
[3]Georgia Institute of Technology, Atlanta, USA
[4]University of Tennessee, Knoxville, USA

Abstract

With the recent advent of many-core architectures such as chip multiprocessors (CMP), the number of processing units accessing a global shared memory is constantly increasing. Co-scheduling techniques are used to improve application throughput on such architectures, but sharing resources often generates critical interferences. In this paper, we focus on the interferences in the last level of cache (LLC) and use the *Cache Allocation Technology* (CAT) recently provided by Intel to partition the LLC and give each co-scheduled application their own cache area. We consider m iterative HPC applications running concurrently and answer to the following questions: (i) how to precisely model the behavior of these applications on the cache partitioned platform? and (ii) how many cores and cache fractions should be assigned to each application to maximize the platform efficiency? Here, platform efficiency is defined as maximizing the performance either globally, or as guaranteeing a fixed ratio of iterations per second for each application. Through extensive experiments using CAT, we demonstrate the impact of cache partitioning when multiple HPC application are co-scheduled onto CMP platforms.

Keywords: Co-scheduling, cache partitioning, HPC application, chip multiprocessor (CMP).

1 Introduction

Co-scheduling applications on a chip multiprocessor (CMP) has received a lot of attention recently ([MSM+11, LCG+16]). The main motivation is to improve the efficiency of the parallel execution of each application. Consider for instance the Gyoukou ZettaScaler supercomputer, currently ranked #4 in the TOP500 benchmark ([Eri17]): it uses PEZY-SC2, a 2048-core processor chip sharing a 40MB last level cache (LLC) ([Com17]): with so many cores at disposal, few applications can efficiently be deployed on the entire computing platform. This is because most application speedup profiles obey Amdahl's law, which tends to severely limit the number of cores to be used in practice.

The remedy is simple: schedule many applications to execute concurrently; each application receives only a fraction of the total number of cores, hence its parallel efficiency is improved. The fraction of computing resources that should actually be assigned to each application depends on many factors, including speedup profiles, but also external constraints prescribed by the user, such as response times or application priorities.

A preliminary version of this work appeared in the proceedings of IEEE Cluster 2018.

Unfortunately, the remedy comes with complications: when multiple applications run concurrently on a CMP, they compete to access shared resources such as the LLC, and their performance actually degrades. This issue turned out so severe ([LK14, ZBF10]) that the name *co-run degradation* has been coined. Modeling and studying cache interferences to prevent co-run degradation has been the object of many studies ([ZLMT14, BCSM08, TJS09]) (see Section 2 on related work for more details).

Intel recently introduced a new hardware feature for cache partitioning called *Cache Allocation Technology* (CAT) ([Ngu16]). CAT allows the programmer to reserve cache subsections, so that when several applications execute concurrently, each of them has its own cache area. Using CAT, Lo et al. ([LCG+16]) showed experimentally that important gains could be reached by co-scheduling latency-sensitive applications with a strict cache partitioning. In this paper, we also use CAT to partition the LLC into several areas when co-scheduling applications, but with the objective of optimizing the throughput of *in-situ* or *in-transit* analysis for large-scale simulations. Indeed, in such simulations, data is generated at each iteration and periodically analyzed by parallel processes on dedicated nodes, concurrently of the main simulation ([S+15]). If these dedicated nodes belong to the main simulation platform (thereby reducing the number of available cores for simulation), we speak of *in-situ* processing, while if they belong to an auxiliary platform, we speak of of *in-transit* processing ([BAA+16]). In both cases, several applications (various kernels for analysis) have to run concurrently to analyze the data in parallel with the current simulation step. The constraint is to achieve a prescribed throughput for each application, because the outcome of the analysis drives the next steps of the simulation. In the simplest case, each application will have to complete within the time of a simulation step, hence we need to achieve the same throughput for each application, and maximize that value. In other situations, some applications may be needed only every k simulation steps, with a different value of k per application ([MVM+15]). This calls for achieving a weighted throughput per application, and for maximizing the minimum value of these weighted throughputs, which dictates the global rate at which the analysis can progress.

The first major contribution of this paper is to introduce a model that characterizes application performance, and to show how to optimally decide how many cores and which cache fraction should be assigned to each application in order to maximize the weighted throughput. The second major contribution is to provide an extensive set of experiments conducted on the Intel Xeon, which assesses the gains achieved by our optimal resource allocation strategy.

The rest of the paper is organized as follows. Section 2 provides an overview of related work. Section 3 details the main framework and all application/platform parameters, as well as the optimization problem. Section 4 presents five co-scheduling strategies, including a dynamic programming approach that provides an optimal resource assignment (according to the model). Section 5 describes the real cache partitioned platform used to perform the experiments. Section 6 assesses the accuracy of the model. Section 7 reports extensive experiments. Finally, Section 8 summarizes our main contributions and discusses directions for future work.

2 Related work

Recent multi-core processors show dozens of cores and large shared caches. In this context, co-scheduling has been extensively studied ([MSM+11, LCG+16]). The main idea behind co-scheduling is to execute applications concurrently rather than in sequence in order to improve the global throughput of the platform. Indeed, many HPC applications are not perfectly parallel, and it is not beneficial to deploy them on the entire platform: the application speedup becomes too low beyond a given core count. A new trend in large-scale simulations are *in-situ* and *in-transit* approaches, to visualize and analyze the data during the simulation ([DR14]). Basically, the idea behind these approaches is that a new dataset is generated periodically, and we need to run different applications on different parts of this dataset before the next period. In the *in-situ* approach, simulation and analysis are co-located in the same node, while in the *in-transit* approach, the data analysis are outsourced onto dedicated nodes ([BAA+16]). Several studies have shown that large-scale simulations with *in-situ* could benefit from co-scheduling approaches ([S+15, BHP+17]). The difficulty consists in ensuring that the in-situ part processes the data fast enough to avoid slowing down the main simulation, which is directly related to co-scheduling issues: how to partition the resources across the concurrent analysis applications that share the CMP?

Shared resources include cache, memory, I/O channels and network links, but among potential degradation factors, cache accesses are prominent ([ZSB+12]). Modeling application interferences is

challenging, and one way to address this problem is to partition the cache to avoid these potential interferences. Multiple cache partitioning schemes have been designed, through hardware techniques ([KCS04, QP06, NLS07]) and software techniques ([TDF90, TASS07, LLD+08, GSYY09]). Most of the hardware approaches are efficient with a very low overhead at the execution time, but they suffer from an extra cost in terms of hardware components. In addition, hardware solutions are difficult to implement and often only tested through simulated architectures. On the side of software-based solutions, the most popular is *page coloring*, where physical pages are selected for application allocations so that they end up in specific sections of the cache. [TASS07] showed that important gains can be achieved through a static partitioning of the L2 cache using page coloring. Besides static strategies, dynamic cache partitioning strategies using page coloring have also been studied. In [LLD+08], the cache partitioning is refined and adjusted periodically at runtime, with the objective to maximize platform efficiency.

Modeling application interference is a challenging task. Hence, [HSPE08] showed, with the Power Law of cache misses (or the $\sqrt{2}$ rule), how the cache size affects the cache miss ratio. The Power Law states that, if for a baseline cache of size C_0, the cache miss ratio is equal to m_0, then for a cache of size C, the cache miss ratio $m = m_0 \left(\frac{C_0}{C}\right)^{\alpha}$, where α is usually set to 0.5.

In a previous work ([ABD+18]) using this law, we were focusing on a static allocation of LLC cache fractions, and core numbers, to concurrent applications as a function of several parameters (cache-miss ratio, access frequency, operation count). We used simulations to assess the performance of our algorithms, because at that time no cache partitioning technologies were available. Furthermore, we were only considering the makespan of the co-schedule, while we aim here at maximizing a weighted throughput. Indeed, this new objective better fits the target applications that we execute on the platform. Also, we focus on iterative HPC kernels, instead of general applications obeying Amdahl's law as in [ABD+18]. Finally, we focus exclusively on integer numbers of cache fractions and processors, since fractions cannot be assigned on the Intel Xeon.

Intel recently released a new software technique to internally partition the last level cache (LLC), called the *Cache Allocation Technology* (CAT) ([Ngu16, LCG+16]). In this paper, we use CAT to experiment with a real cache partitioned platform. To the best of our knowledge, this work is the first co-scheduling study for a cache partitioned system (using CAT) with HPC workloads.

3 Model and optimization problem

In this section, we first describe the application model, and then we formalize the optimization problem.

3.1 Application model

The objective is to execute m iterative applications $A_1, \ldots, A_m$ on P identical cores. The applications are sharing a cache of size C. As explained in Section 1, new technologies allow us to decide how many cores and which fraction of cache are allocated to each application. Specifically, the cache can be divided into X different fractions. For instance, if $X = 20$, we can give several fractions of 5% of the cache to each application.

Let p_i be the number of cores on which application A_i is executed, and let x_i be the number of fractions of cache assigned to A_i, for $1 \leq i \leq m$. Hence, A_i uses a cache of size $\frac{x_i}{X}C$. We must have $\sum_{i=1}^{m} p_i = P$ and $\sum_{i=1}^{m} x_i = X$, i.e., all the cores and the cache fractions are partitioned across the applications.

Given p_i and x_i, an application A_i executes one iteration in time

$$T_i(p_i, x_i) = t_i(p_i)\left(1 + h_i(x_i)\right), \tag{1}$$

where $t_i(p_i)$ represents the computation cost and $h_i(x_i)$ the slowdown induced by cache misses in the LLC. Intuitively, the computation cost decreases when p_i increases, and similarly, the slowdown decreases when x_i increases. In this formula, the slowdown incurred by cache misses does not depend on the number of cores assigned to the application. We keep this assumption in our model, and discuss its accuracy in Section 6, where we measure cache misses and refine the model.

Assumption 1. *In the execution time, the slowdown due to cache misses does not depend on the number of cores involved.*

We now detail the model for $t_i(p_i)$ and $h_i(x_i)$.

Computations $t_i(p_i)$. We assume that all applications obey Amdahl's law ([Amd67]), hence

$$t_i(p_i) = s_i T_i^{seq} + (1 - s_i)\frac{T_i^{seq}}{p_i}, \tag{2}$$

where T_i^{seq} is the sequential time of the application executed with 100% of the cache, and s_i is the sequential fraction of the application.

Cache misses effect $h_i(x_i)$. The most challenging part is to model the slowdown factor $h_i(x_i)$. In chip multiprocessors (CMP), many studies have observed that cache miss ratio follows the Power Law, also called the $\sqrt{2}$ rule ([HSPE08, KSS12, RKB+09]). The Power Law of cache misses states that for a cache of size C_{act}, the cache miss ratio r can be expressed as

$$r = r_0 \left(\frac{C_0}{C_{act}}\right)^{\alpha}, \tag{3}$$

where r_0 represents the cache miss ratio for a baseline cache of size C_0, and α is a parameter ranging from 0.3 to 0.7, with an average at 0.5. We consider $\alpha = 0.5$ in the following.

We slightly generalize the Power Law formula (with $\alpha = 0.5$) to avoid side effects, and define the slowdown as follows:

$$h_i(x_i) = a_i + \frac{b_i}{\sqrt{x_i}}, \tag{4}$$

where a_i and b_i are constants depending on the application A_i. From Equation (3) with $\alpha = 0.5$, we have $b_i = r_0\sqrt{\frac{C_0 X}{C}}$ (since $C_{act} = \frac{x_i}{X}C$). In Section 6, we determine a_i and b_i by interpolation, from experimentally measured cache misses, see Table 2.

Finally, when assigning p_i cores and a fraction x_i of the cache, an application A_i executes one iteration in time

$$T_i(p_i, x_i) = t_i(p_i)\left(c_i + \frac{b_i}{\sqrt{x_i}}\right), \tag{5}$$

where $c_i = 1 + a_i$.

3.2 Optimization problem

As stated in Section 1, the goal is to maximize a weighted throughput, since analysis applications may be required at different rates, from every simulation step to every tenth (or more) step ([MVM+15]). We let β_i denote the weight of application A_i for $1 \le i \le m$. Intuitively, β_i represents the number of times that we should execute application A_i at each iteration step. These priority values are not absolute but relative: for $m = 2$ applications, having $\beta_1 = \frac{1}{4}$ and $\beta_2 = 1$ means we execute four times A_2 (at each step) while executing A_1 only once (every fourth step). This is equivalent to having $\beta_1 = 1$ and $\beta_2 = 4$ if we change the granularity of the simulation steps. In fact, what matters is the relative number of executions of each A_i that is required, hence we aim at maximizing the weighted throughput:

- The throughput achieved when executing β_i instances of application A_i is $\frac{1}{\beta_i T_i(p_i,x_i)}$;
- The objective is to partition the shared cache and assign cores such that the total time taken by the slowest application is minimal, i.e., the lowest weighted throughput is maximal.

The weighted throughput allows us to ensure some fairness between applications, and to enforce a better analysis rate of the simulation results whenever the bottleneck is the slowest application. Of course, letting $\beta_i = 1$ lead to maximizing the rate of the analysis when all applications are needed at the same frequency. The optimization problem is formally expressed below:

Definition 1 (CoSched-CachePart). *Given m iterative applications with priorities $(A_1, \beta_1), \ldots, (A_m, \beta_m)$ and a platform with P identical cores sharing a memory of size C with X fractions of cache, the*

CoSched-CachePart *problem consists in finding a schedule* $\{(p_1, x_1), \ldots, (p_m, x_m)\}$ *such that*

$$\text{Maximize} \quad \min_{1 \leq i \leq m} \left\{ \frac{1}{\beta_i T_i(p_i, x_i)} \right\}$$

$$\text{subject to} \quad \begin{cases} \sum_{i=1}^{m} p_i = P, \\ \sum_{i=1}^{m} x_i = X. \end{cases}$$

4 Scheduling strategies

In this section, we introduce several co-scheduling strategies that we will compare via experiments on the Intel Xeon. We start with a (theoretically) optimal schedule, and then present simple heuristics that we use for comparison.

4.1 Optimal solution to CoSched-CachePart

Given the time to execute one iteration of application A_i with p_i cores and a fraction x_i of the cache $T_i(p_i, x_i)$, we can solve the CoSched-CachePart problem optimally, with a dynamic programming algorithm.

Theorem 1. CoSched-CachePart *can be solved in time* $O(mPX)$*, where* m *is the number of applications,* P *is the number of processors, and* X *is the number of different possible cache fractions.*

Proof. Let $T(i, q, c)$ be the maximum weighted throughput that can be obtained with applications $A_1, \ldots, A_i$, using q cores and c fractions of cache. The goal is to find $T(m, P, X)$. We compute $T(i, q, c)$ as follows:

$$T(i,q,c) = \begin{cases} \max\limits_{\substack{1 \leq q_1 \leq q \\ 1 \leq c_1 \leq c}} \frac{1}{\beta_1 T_1(q_1, c_1)} & \text{if } i = 1, \\ \max\limits_{\substack{1 \leq q_i < q \\ 1 \leq c_i < c}} \left\{ \min \left\{ T(i-1, q-q_i, c-c_i), \right. \right. & \\ \qquad \left. \left. \frac{1}{\beta_i T_i(q_i, c_i)} \right\} \right\} & \text{otherwise.} \end{cases}$$

The base case $i = 1$, for one application, takes the best out of all possible allocations (in terms of number of processors and number of cache fractions). Note that for most execution time profile, the execution time in this case is obtained by $T(1, q, c) = \frac{1}{\beta_1 T_1(q,c)}$, since using less processors or less fractions of cache would only increase the execution time, but we write the general expression to encompass any execution time profile, and not only the one given by Equation (5).

In the recurrence, we try all possible number of processors and number of cache fractions for application i, and re-use the optimal solution for the $i-1$ other applications. If we did not use the optimal solution, we would be able to create a better solution, hence it is easy to see that the problem has an optimal substructure property and can be solved with a dynamic programming algorithm.

There are mPX values to compute, and they can each be obtained in constant time, except for the generalized base case, where we need to perform a maximum over PX values. Overall, with the execution profile of our model, we can compute all values in time $O(mPX)$, and the complexity becomes $O(mP^2X^2)$ in the general case. In practice on the Intel Xeon, we have $m \leq P = 14$, and $X = 20$, hence the dynamic programming algorithm executes almost instantaneously in all the experiments. □

This optimal algorithm provides us with our first strategy to schedule applications, and it is called DP-CP (Dynamic Programming with Cache Partitioning). Checking the behavior of this strategy in practice will assess the accuracy of the performance model, when using the values of $T_i(p_i, x_i)$ obtained with the model of Section 3.

4.2 Equal-resource assignment

To evaluate the global efficiency of the optimal solution for DP-CP, we compare it to Eq-CP, a simple strategy that allocates the same number of cores and the same number of cache fractions to each application. The algorithm is the following: we start to give $x_i = \lfloor \frac{X}{m} \rfloor$ and $p_i = \lfloor \frac{P}{m} \rfloor$ for all i, then, we give the $P \mod m$ extra cores one by one to the first $P \mod m$ applications, and we give the $X \mod m$ extra cache fractions one by one to the last $X \mod m$ applications. Doing this, we forbid the case where an application receives an extra core plus an extra fraction of cache, thereby avoiding a totally unbalanced equal assignment.

4.3 Impact of cache allocation

In order to isolate the impact of cache partitioning on performance, we introduce some variants where only the cache allocation is modified:

- DP-Equal uses the number of cores returned by the dynamic programming algorithm, hence the same as for DP-CP, but shares the cache equally across applications, as done by Eq-CP.
- We also consider strategies that do not enforce any cache partitioning, but only decide on the number of cores for each application. DP-NoCP uses the same number of cores as DP-CP, and Eq-NoCP uses an equal-resource assignment as in Eq-CP. However, for these two strategies, all applications share the whole cache, i.e., CAT is disabled.

Algorithm 1: Equal allocation with cache partitioning

```
1 Eq-CP (m, P, X) begin
2 |   for i = 1 to m do p_i ← ⌊P/m⌋; x_i ← ⌊X/m⌋;
3 |   for i = 1 to P mod m do p_i ← p_i + 1 ;
4 |   for i = 1 to X mod m do x_{m+1-i} ← x_{m+1-i} + 1 ;
5 end
```

5 Experimental setup

In this section, we first describe the platform and the benchmark applications in Section 5.1. Then in Section 5.2, we explain in details the *Cache Allocation Technology* CAT.

5.1 Platform and applications

The experimental platform is composed of a Dell PowerEdge R730 server with two Intel Xeon E5-2650L v4 processors (*Broadwell* microarchitecture). Each processor contains $P = 14$ cores (with Hyper-Threading disabled) sharing a 35MB last-level cache (*Cluster-on-Die* disabled), divided into $X = 20$ slices (or fractions). Nodes run a vanilla 4.11.0 Linux kernel with cache partitioning enabled.

Only one processor (with 14 cores) is used for the experiments, since the LLC is not shared across processors. It matches standard practice because users who co-schedule real-applications often place each application inside a single processor to benefit from the shared cache. Batch schedulers also allocate cores of the same processor whenever possible. Hence our work focuses on co-scheduling the subset of applications that are assigned to a single processor by the user or by the batch scheduler.

Cache experiments are very sensitive to perturbations, so we take great care to ensure that all experiments are fully reproducible. To avoid perturbations: (i) we average values obtained (like cache misses) over 20 (in Section 6) or 5 (in Section 7) identical runs; (ii) we flush the last-level cache entirely between runs; and (iii) experiments run on a dedicated processor while the program launching and monitoring them runs on the other processor. All the data presented in this paper (cache misses, number of floating operations, etc), is obtained with PAPI 5.5.1 ([BDG+00]). Each benchmark is compiled using the Intel Fortran Compiler 17.0.1 with the optimization level `-O2` and the flag `-mcmodel=medium`.

For validations and performance evaluation, we use six HPC workloads from the NAS benchmarks ([B+91]) (see Table 1). We consider only NAS benchmarks from class A, as detailed in Table 1.

This benchmark allows us to compare different type of applications, ranging from compute-intensive to memory-intensive kernels. We have tried most combinations of applications, in particular using CG,

App	Description
CG	Uses conjugate gradients method to solve a large sparse symmetric positive definite system of linear equations
BT	Solves multiple, independent systems of block tridiagonal equations with a predefined block size
LU	Solves regular sparse upper and lower triangular systems
SP	Solves multiple, independent systems of scalar pentadiagonal equations
MG	Performs a multi-grid solve on a sequence of meshes
FT	Performs discrete 3D fast Fourier Transform

Tab. 1: Description of the NAS parallel benchmarks.

FT and MG since they lead to significant results. We believe that the heuristics should behave similarly on other applications, and the interested reader is encouraged to experiment with its own applications. The code of our heuristics is available at `graal.ens-lyon.fr/~abenoit/code/cachepart.tgz`.

5.2 Cache Allocation Technology

The Cache Allocation Technology (CAT) ([Ngu16]) is part of a larger set of Intel technologies that are called the Resource Director Technology (RDT) and supported since the *Haswell* architecture. RDT lets the operating system group applications into classes of service (COS). Each class of service describes the amount of resources, in particular cache, that assigned applications can use.

The CAT divides the LLC into X slices of cache (see Figure 1). Each class of service has a set of slices that applications can use: When reading or writing memory requires to fetch a cache line in the LLC, that cache line must be allocated in the slices available to the class of the current application. However applications may read/modify cache lines that are already available in other slices, for instance when sharing memory between programs in different classes (each cache line can only exist once in the entire cache).

Each slice may only be used by a single class. By default, applications are placed in the default class (COS_0) which contains slices not used by any other class. The set of slices available to a class is a capacity bit-mask (CBM) of length X. With $X = 20$, if COS_1 has access to the last 4 slices (the top 20% of the LLC), CBM_1 would be set to `0xf0000`.

Note that CAT has some technical restrictions:

- The number of slices (CBM length) and classes are architecture dependent (20 and 16 on our platform);
- A CBM cannot be empty (each class of applications must have at least one fraction of cache);
- Bits set in a CBM must be contiguous;
- Slices are not distributed geographically in the LLC, and address hashing ensures spreading of slices over the entire LLC; In other words, `0x10000` and `0x00001` CBM should behave exactly the same with respect to locality; there are no NUCA effects (Non Uniform Cache Access).

Also, we consider a strict cache partitioning, hence each COS contains only one application (and each cache slice is available to a single application).

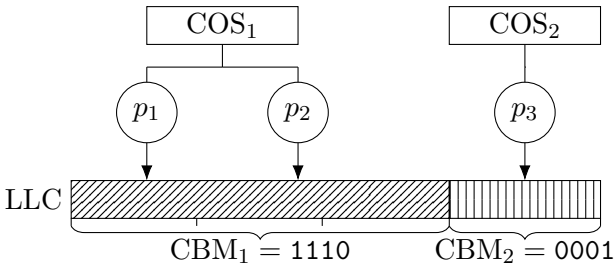


Fig. 1: CAT example with 2 classes of service, 3 cores and a 4-bit capacity mask (CBM). First COS has 2 cores and 75% of the LLC, the second class of service has the remaining resources.

6 Accuracy of the model

In this section, we assess the precision of the model developed in Section 3. First, we detail the experimental protocol and explain how to obtain the model parameters for each application in Section 6.1. Then, we study in Section 6.2 the behavior of cache misses on the platform described in Section 5.1, so as to verify whether the Power Law holds for HPC workloads on such architectures. Finally, we study in Section 6.3 the accuracy of the model proposed in Section 3.1 by comparing the expected execution time from Equation (5) to the measured one.

6.1 Experimental protocol

To instantiate the model and check its accuracy, we need to find for each application the value of three parameters used in Equation (5): s_i (sequential fraction), a_i (or equivalently $c_i = a_i + 1$), and b_i (cache slowdown). To this purpose, we monitor each application with PAPI ([BDG+00]) and use multiple interpolations on the produced data to find the desired constants. More precisely, we proceed as follows. Each application A_i executes alone on a dedicated processor. First, we give 100% of the cache to the application A_i and vary the number of cores from 1 to 14 to derive the sequential fraction s_i. Then, for each cache fraction x_i ranging from 15% to 85%, we record the number of cache misses when p_i ranges from 1 to 14 and derive values for c_i and b_i. Finally, we put the pieces together, keeping the value of s_i while scaling c_i and b_i by a constant factor, thereby deriving the final values for $T_i(p_i, x_i)$ in Equation (5).

As a side note, we point out that this complicated (and definitely not scalable) approach was necessary because the least-square interpolation program would not converge when fed directly with 80% of the 280 experimental values for each application (14 processors, and 16 values of x out of 20). We expect it will be even more challenging to instantiate the model for future platforms where the number of cores will be higher.

Note that the Power Law with $\alpha = 0.5$ suits well the behavior of compute-intensive benchmarks such as CG, but struggles to model memory/communication-intensive applications such as MG and FT. The results for all applications are displayed in Table 2.

6.2 Accuracy of the Power Law

Figure 2 shows the evolution of cache miss ratios for the six applications depending on the number of cores and cache fraction. We observe that for most applications, the cache miss ratio increases with the number of cores for small cache fractions, while it does not vary significantly with the number of cores for higher cache fractions. Therefore, these results verify the assumption about the relation between number of cores and cache misses (Assumption 1).

On Figure 3, we study the evolution of cache miss ratios for each considered application, running alone with a single core. We do not look at cache fractions below $x = 3$ (or 15%) because, according to our experiments, it shows irrelevant results due to cache contention. We observe that the Power Law with $\alpha = 0.5$ suits well the behavior of compute-intensive benchmarks CG, BT, LU and SP, but struggles to model memory/communication-intensive applications like MG and FT.

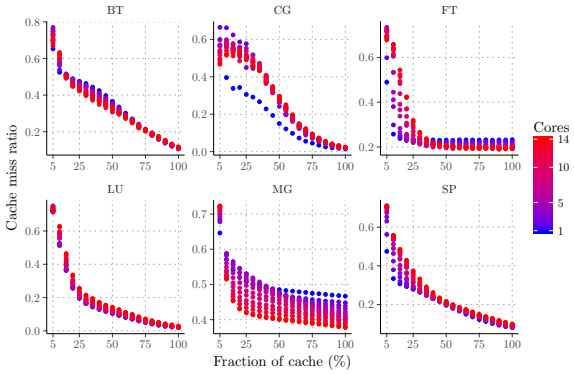


Fig. 2: Evolution of cache miss ratio when the cache fraction x_i is ranging from 1 to 20 (i.e., from 5% to 100%) and the number of cores p_i is ranging from 1 (blue) to 14 (red).

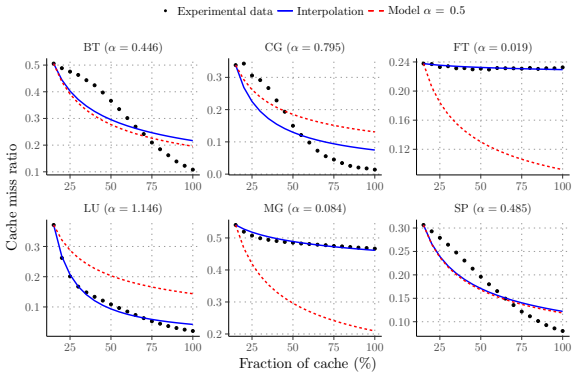


Fig. 3: Comparison between the predicted cache miss ratio given by the Power Law with $\alpha = 0.5$ in red, the best found α parameter in blue and the measured cache miss ratio in black. Applications run alone on the platform with 1 core.

App_i	a_i	b_i	s_i
BT	-0.0026	0.0287	0.010
CG	-0.0379	0.0474	0
FT	0.0092	0.0129	0.016
LU	-0.0247	0.0275	0.020
MG	0.0460	0.0073	0.065
SP	-0.0110	0.0254	0.018

Tab. 2: s_i, a_i and b_i obtained by interpolation from the data produced by measurements (averaged on the core numbers, according to Assumption 1).

6.3 Accuracy of the execution time

We aim at verifying the accuracy of the execution time predicted by the model. Figure 4 shows, for each application, the comparison between the measured execution time and the model, when the application runs alone on the platform (no co-scheduling here). In Figure 4 , the number of cores varies from 1 to 14 while the cache fraction is fixed at $x = 3$ (or 15%).

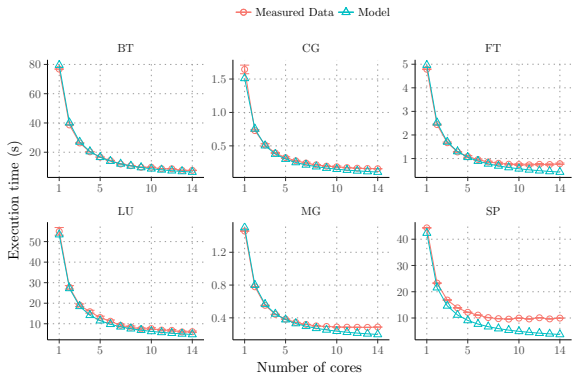


Fig. 4: Comparison between predicted execution time by the model and measured execution time, when varying the number of cores up to 14 and with a cache fraction set to 15%.

Figure 5 shows the relative error between predictions and the real data. The relative error is defined as

$$E_i(p_i, x_i) = \frac{\left|T_i(p_i, x_i) - T_i^{real}(p_i, x_i)\right|}{T_i^{real}(p_i, x_i)},$$

where $T_i^{real}(p_i, x_i)$ is the measured execution time on the cache partitioned platform for application A_i with p_i cores and x_i fractions of cache. We observe that our model predicts execution times rather well for LU, BT, CG and MG, with less than 25% of error for worst cases. For FT, the model is accurate for $x_i \geq 6$ (30%) and $p_i \leq 10$, with a relative error below 15%, but the model loses accuracy for small cache fractions and high number of cores. For SP, we have the same observation, the model is not accurate for a number of cores larger than 8 if the cache fraction is below 50% (the red part in the Figure 5). This is due to a specific behavior of FT and SP: their execution times tend to become constant after a certain core threshold (see Figure 4), while the model expects a strictly decreasing execution time. For both applications, this constant plateau is not due to Amdahl's law (both FT and SP are parallel enough to scale up to 14 cores), hence a contention effect (either from the cache or the memory bandwidth) is probably behind this constant level in performance. Another reason to explain these mis-predictions when the number of cores increases, is Assumption 1, which states that the number of cores does not impact LLC cache misses, which is not true for all applications in practice.

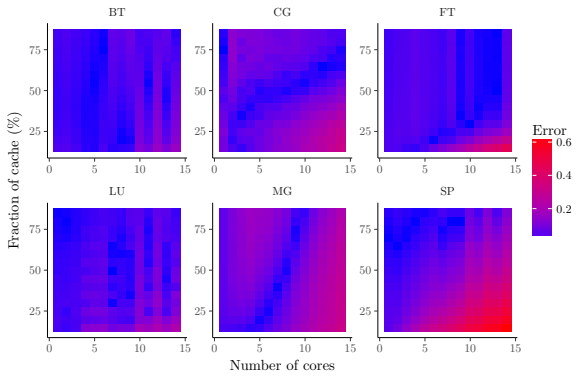


Fig. 5: Heat-map of the relative error between the model predictions and the measured execution times when the cache fraction is varying from 15% to 85% and the number of cores from 1 to 14.

7 Results

To assess the performance of the scheduling strategies of Section 4 and to evaluate the impact of cache partitioning on co-scheduling performance, we conduct an extensive campaign of experiments using a real cache partitioned system.

7.1 Experimental protocol

The platform and the applications used for all the experiments are described in Section 5. Recall that we consider iterative applications, hence we have modified their main loop such that each of them computes for a duration T. We choose a value for T large enough to ensure that each application reaches the steady state with enough iterations (for instance, $T = 3$ minutes for small applications like CG, FT, MG and $T = 10$ minutes for the others). If a co-schedule contains both small and big applications, we use $T = 10$ minutes for all applications. In addition, for all the following experiments, we use 12 cores out of the 14 available, to avoid rounding effects when we co-schedule a number of applications that is not divisible by the number of cores.

Evaluation framework. To study the performance of the different algorithms in terms of weighted throughput, we measure the time for one iteration of A_i: $T_i = \frac{T}{\#\texttt{iter}_i}$, where $\#\texttt{iter}_i$ is the number of iterations of application A_i during T. Then, we compute $\min_i \frac{1}{\beta_i T_i}$. We are then interested in the relative speed of each application with respect to the others. Indeed, recall that for all i, j, the goal is to have $\beta_i T_i = \beta_j T_j$, by definition of the β's. Hence, we further study the following fairness criterion, representing the distance to the optimal fairness, $\Delta_{fairness}$:

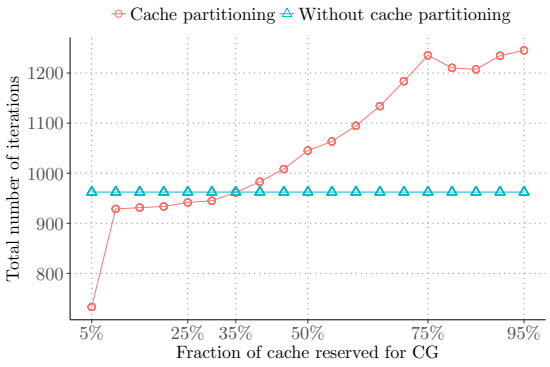
$$\Delta_{fairness} = \sum_{i \neq j} \left| \frac{\beta_i T_i}{\beta_j T_j} - 1 \right|. \tag{6}$$

In addition to studying the maximum weighted throughput that can be obtained with the applications, we also report the value of $\Delta_{fairness}$ in the experiments, so as to assess whether the heuristics are ensuring that the correct number of iterations of each application is performed during a given amount of time. The goal is to have $\Delta_{fairness}$ as close to 0 as possible.

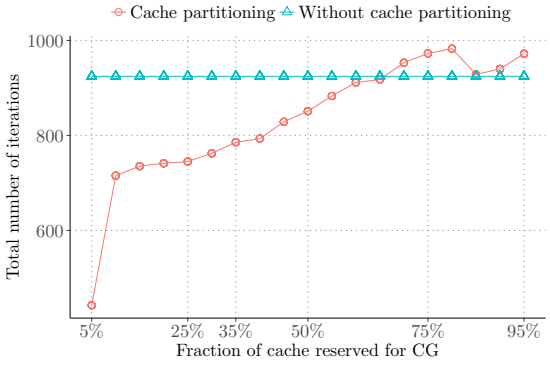
7.2 Impact of cache partitioning

The first step is to assess the impact of cache partitioning (CP) on performance. To this purpose, we co-schedule two applications, so we have three combinations (CG+MG, CG+FT, FT+MG). For all i, j, we set the number of cores for A_i and A_j to six, and we vary the fraction of cache allocated to A_i from 5% to 95% while, at the same time, the cache fraction of A_j is varying from 95% to 5%. The y-axis represents

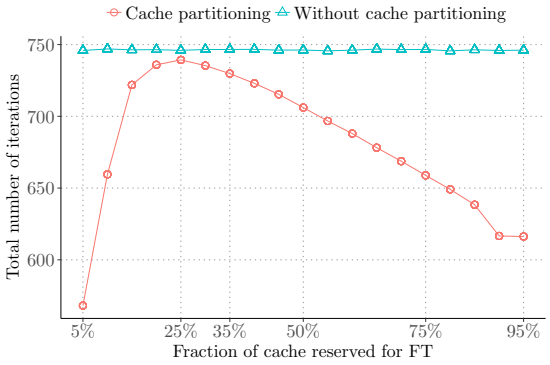
the aggregated number of iterations executed by all applications. We run the applications both with CP enabled, and CP not enabled. Figure 6a shows the impact of CP for CG+MG: we can see that when CG has more than 35% of the cache, CP outperforms the version without CP. The impact of CP lies in the behavior of each application, more specifically their data access pattern. CG is a compute intensive application with an irregular memory access pattern, while MG is a memory intensive application. More specifically, MG does not take a great benefit for more cache after 35%, while the performance of CG greatly depends on the cache size (for more details on application behaviors, see Figure 2). Without a cache partitioning scheme, by reading/writing a lot of different cache lines, MG will often evict CG cache lines, resulting into a performance degradation of both applications.



(a) CG and MG



(b) CG and FT



(c) FT and MG

Fig. 6: CG+MG, CG+FT or FT+MG with 6 cores for each application, with different cache partitioning strategies.

Figure 6b shows the impact of CP for CG+FT. In this case, we note a small improvement when CG has 80% of the cache. The reason behind this improvement is that FT is more communication intensive (all-to-all communication) than strictly memory intensive, hence the gain obtained by CP is

less important than for CG+MG. Since we consider only one processor, the applications that run are the shared memory version (OpenMP), and in that context, the impact of cache on communications is small.

Finally, Figure 6c presents the result for the last combination FT+MG. The cache partitioning is not efficient for that combination of two memory and communication intensive applications. If FT has 25% and MG has 75%, then CP can almost achieve the same performance as without CP. This inefficiency is mostly due to the memory intensive and communication intensive behaviors of both applications involved, none of them needs a strict cache partitioning, since their use of the cache varies during iterations.

Summary. The cache partitioning is very interesting when compute-intensive and memory-intensive application are co-scheduled (important gain, up to 25%, for CG+MG, small gain for CG+FT). On the contrary, FT and MG together perform badly with the cache partitioning enabled, these applications do not benefit from the cache to improve their execution time by iteration. Hence, the behavior of applications has a strong impact on the global performance of cache partitioning, and in general, co-scheduling applications with the same behavior results in degraded global performance when using CP.

7.3 Co-scheduling results with two applications

Now that we have demonstrated the interest of cache partitioning, we study the performance of the scheduling strategies of Section 4. Recall that the CoSched-CachePart optimization problem aims at maximizing the minimum weighted throughput among co-scheduled applications. Considering two applications (A_i, A_j), for β_i iterations of A_i, we aim at performing β_j iterations of A_j. To avoid some cache effects that appear when the cache area is too small, we set the minimum cache fraction allocated to each application to three (each application has at least 15% of the cache), while the minimum number of cores per application is set to one. We use three different ways to present the result for each studied combination: (i) the objective we want to maximize (minimum weighted throughput), (ii) the ratio of iterations done, and (iii) the fairness $\Delta_{fairness}$ defined in Equation (6).

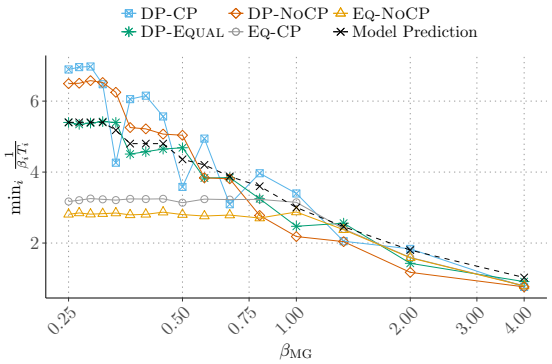
CG+MG. On Figure 7a, we see what is the minimum throughput achieved by each method for CG+MG. The weight β associated to MG varies from 0.25 to 4. The algorithms based on dynamic programming DP-CP, DP-Equal and DP-NoCP outperform both equal-resource assignment heuristics Eq-CP and Eq-NoCP. In this scenario, the cache partitioning provides a good performance improvement, since on average DP-CP outperforms DP-NoCP.

Figure 7b shows the ratio of iterations for CG+MG. Ideally, we would like to obtain $\beta_{\text{CG}}T_{\text{CG}} = \beta_{\text{MG}}T_{\text{MG}}$, the dashed black line represents that optimal iteration ratio. First, note that Eq-CP and Eq-NoCP show constant results because they do not depend on weight, but Eq-CP performs better (even without a clever algorithm, cache partitioning helps). Second, we observe that DP-CP is the closest (on average) to the ideal line, hence the cache partitioning really helps here.

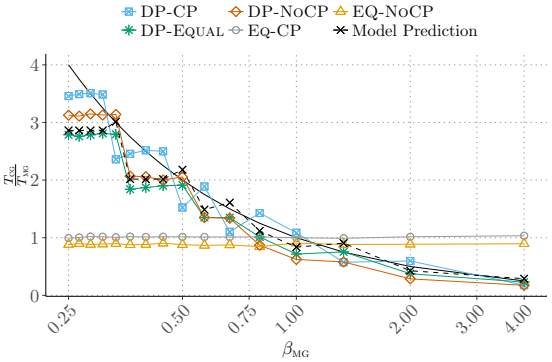
Finally, Figure 7c presents the fairness $\Delta_{fairness}$, as defined in Equation (6). We observe that DP-CP, DP-NoCP and DP-Equal exhibit the same $\Delta_{fairness}$, near to zero, while Eq-CP and Eq-NoCP are far from the optimal fairness.

CG+FT. In Figure 8a, we observe that DP-CP, DP-Equal and DP-NoCP outperform Eq-CP and Eq-NoCP when β_{FT} is larger than 0.5. Only, DP-NoCP outperforms Eq-NoCP all the time. When β_{FT} is smaller than 0.5, the two variants without cache partitioning perform better than the two versions with cache partitioning. As explained in Section 7.2, due to its communication-intensive behavior, FT will not benefit a lot from cache partitioning techniques. Figure 8b presents the iteration ratio (i.e., the fairness among co-scheduled applications) when we co-schedule CG+FT: DP-CP, DP-Equal and DP-NoCP exhibit good performance, and we are very close to the dashed line that represents the ideal iteration ratio to reach. On Figure 8c, we observe the fairness $\Delta_{fairness}$: Eq-CP and Eq-NoCP show a poor $\Delta_{fairness}$ as expected, and DP-CP, DP-Equal and DP-NoCP show the same good performance, very close to zero.

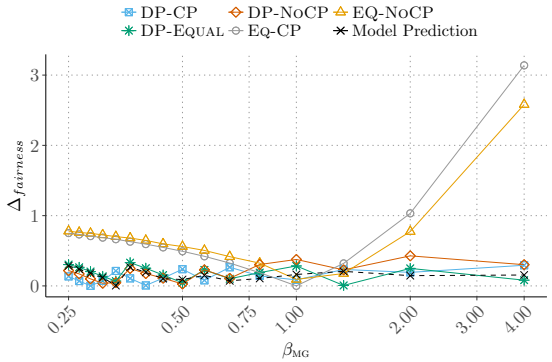
MG+FT. Figure 9a presents the results obtained for MG+FT. DP-CP, DP-Equal and DP-NoCP outperform Eq-CP and Eq-NoCP, except for β_{FT} lower than 0.50. For both DP-CP and Eq-CP, the



(a) Minimum throughput (higher is better).



(b) Iteration ratio done (closer to the solid black line is better).

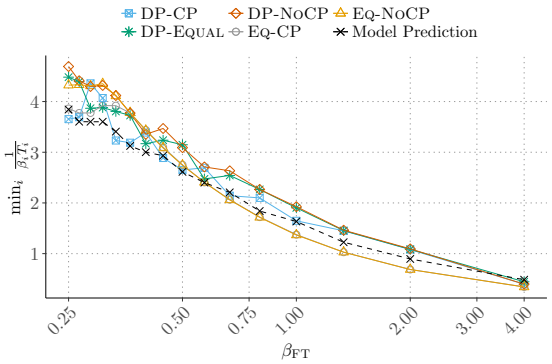


(c) Fairness $\Delta_{fairness}$ (lower is better).

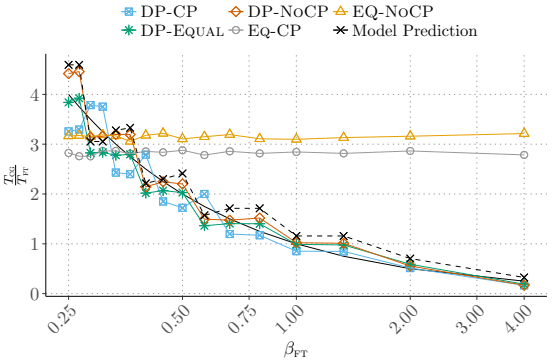
Fig. 7: CG and MG when β_{MG} is varying from 0.25 to 4.

cache partitioning does not bring much improvement. The main reason is that co-scheduling one memory and one communication intensive application is not very efficient (see Section 7.2). Figure 9b shows that DP-CP, DP-Equal and DP-NoCP perform well, very close to the ideal iteration ratio (the dashed line). On Figure 9c, we note that the fairness $\Delta_{fairness}$ is close to zero for DP-CP, DP-Equal and DP-NoCP, while (logically) the $\Delta_{fairness}$ is larger (hence worst) for Eq-CP and Eq-NoCP.

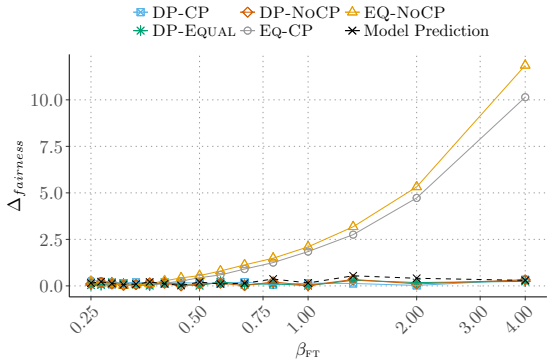
BT, LU, SP co-scheduled with MG. Figures 10, 11 and 12 show the minimum throughput (on the left) and the fairness $\Delta_{fairness}$ (on the right) obtained by co-scheduling, respectively, BT+MG, LU+MG and SP+MG. For the minimum throughput (on the left of each figure), all results are quite similar, and all variants based on our algorithm DP-CP outperform Eq-CP and Eq-NoCP. The cache partitioning does not bring a significant gain in this scenario, but DP-CP is almost always better than DP-NoCP. We observe that DP-Equal always performs worst than DP-CP and DP-NoCP, which means that doing



(a) Minimum throughput (higher is better).



(b) Iteration ratio done (closer to the solid black line is better).

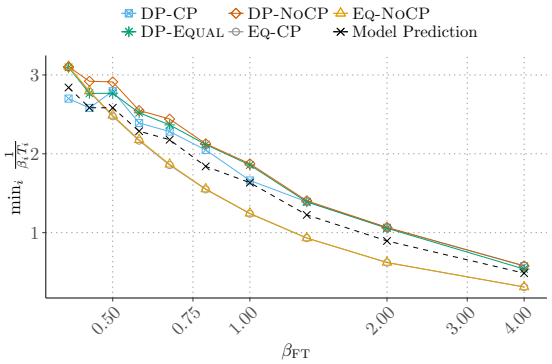


(c) Fairness $\Delta_{fairness}$ (lower is better).

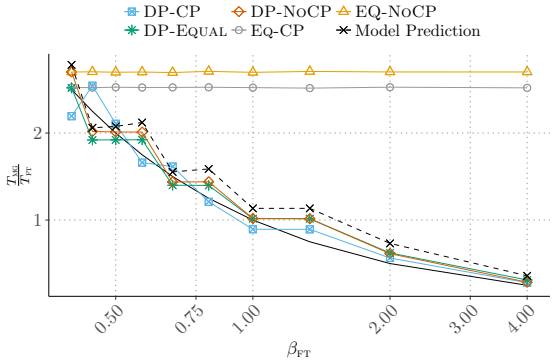
Fig. 8: CG and FT when β_{FT} is varying from 0.25 to 4.

a naive cache partitioning (an equal one in that case) can lead to significant performance degradations. Concerning the fairness $\Delta_{fairness}$, values are quite high in all cases. Indeed, BT, LU and SP are much larger than MG in terms of number of operations (by roughly 10^3), hence it is impossible to do, for instance, four times more iterations of LU than iterations of MG without a very large value of T (the time during which we compute applications).

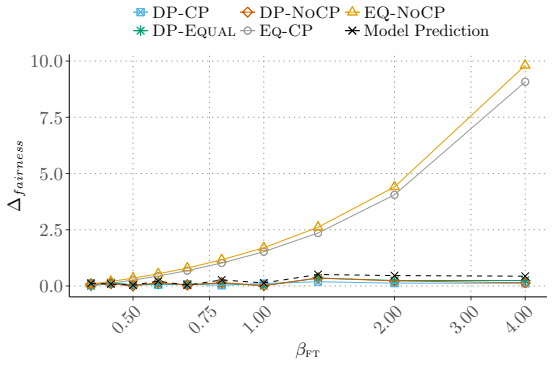
Special case: CG and MG when each application has six cores. We are now considering a special case, in order to investigate how the cache is impacting co-scheduling performance. In this case, all applications have the same number of cores (six in our case), so only the cache is available to increase performance. Figure 13a shows the global performance of all methods. Obviously, only DP-CP takes advantage of this scenario because only this method can choose how to partition the cache. If β_{MG} is smaller than 1, it means that we have to compute more CG than MG, and in that case, the cache has a



(a) Minimum throughput (higher is better).



(b) Iteration ratio done (closer to the solid black line is better).



(c) Fairness $\Delta_{fairness}$ (lower is better).

Fig. 9: MG and FT when β_{FT} is varying from 0.25 to 4.

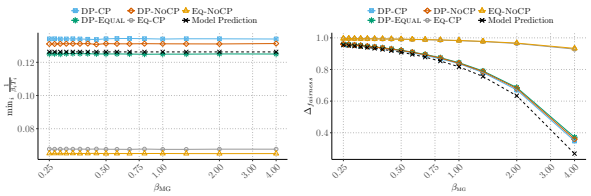


Fig. 10: Minimum throughput and $\Delta_{fairness}$ for BT+MG.

strong effect (up to 25% improvement with cache partitioning enabled). With this scenario, we are able to isolate which part of performance relies on cache effect. Figure 13b depicts the iteration ratio achieved with an equal number of cores for each application. We observe that with only the cache, it is hard to

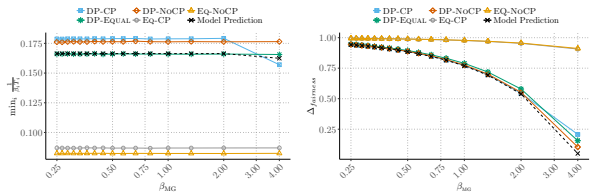


Fig. 11: Minimum throughput and $\Delta_{fairness}$ for LU+MG.

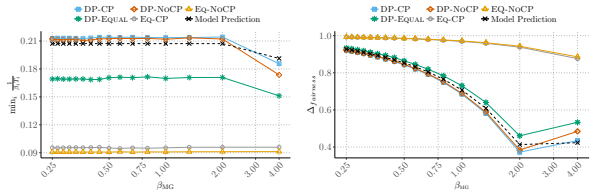


Fig. 12: Minimum throughput and $\Delta_{fairness}$ for SP+MG.

enforce the required ratio of the number of iterations, according to the values of the β_i. Figure 13c represents the fairness $\Delta_{fairness}$ between the ideal iteration ratio and the iteration ratio obtained with each method. Note that the fairness $\Delta_{fairness}$ is high for every method, but the $\Delta_{fairness}$ of DP-CP is the smallest.

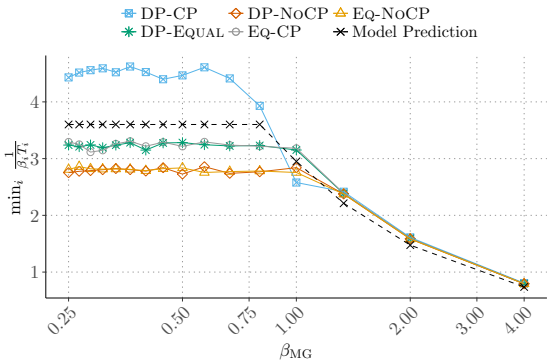
Summary. The model is accurate enough to enforce that the corresponding optimal DP algorithm performs well: in most cases, DP-CP, DP-Equal and DP-NoCP outperform Eq-CP and Eq-NoCP. On the cache partitioning side, when co-scheduling CG and MG, the cache partitioning is really interesting to isolate applications that pollute the cache, such as MG. Figure 13a clearly shows the impact of cache on performance when the number of cores is set for each application. In the worst cases, for instance with FT and MG, the cache partitioning does not improve performance, but does not degrade it either.

7.4 Co-scheduling results with three applications

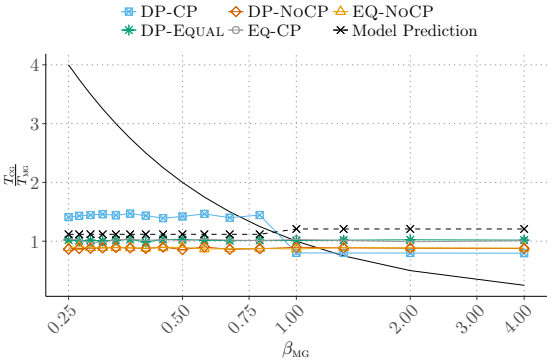
In this section, we present the results with three co-scheduled applications. Similarly to the case with two applications, with three applications (A_1, A_2, A_3), only β_3 is ranging from 0.25 to 4, while $\beta_1 = \beta_2 = 1$. First, we focus only on co-schedules with CG and MG, because they are very interesting applications to study. Second, we study all combinations of co-scheduling with CG, FT and MG. We do not look at the iteration ratio in this section, but focus on minimum throughput and fairness $\Delta_{fairness}$.

2CG+MG. Figure 14 shows the minimum throughput obtained when we co-schedule 2CG+MG, while the weight associated to MG is ranging from 0.25 to 4. Note that it is interesting to co-schedule multiple copies of the same application (two CGs in this scenario) in order to improve the global efficiency, when this application exhibits a speedup profile with limited gain from adding extra cores and/or extra fractions of caches. We observe that the scheduling strategies building on the dynamic programming algorithm DP-CP, DP-Equal and DP-NoCP outperform Eq-CP and Eq-NoCP. In addition, cache partitioning is very helpful in this case: DP-CP exhibits a gain around 15% on average over DP-NoCP and DP-Equal. The fairness $\Delta_{fairness}$ is also depicted on the right. Recall that ideally, we would like to have $\beta_i T_i = \beta_j T_j$ for all i, j (see Equation (6)). We observe that the method that is the closest to zero is DP-CP, confirming the strong influence of cache partitioning.

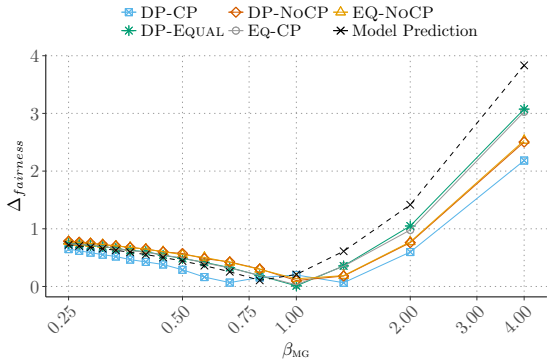
2MG+{CG, BT, LU, SP}. Figure 15 presents the minimal throughput obtained by each method when we co-schedule 2MG+CG, where the weight of CG is ranging from 0.25 to 4. Again, the DP-based strategies DP-CP, DP-Equal and DP-NoCP exhibit good performance for β_{CG} smaller than 0.50, but they suffer from a lack of performance when β_{CG} is between 0.50 and 1. When β_{CG} is larger than 1, DP-CP becomes the best method again. On the right of Figure 15, we can see the confirmation that



(a) Minimum throughput (higher is better).



(b) Iteration ratio done (closer to the solid black line is better).



(c) Fairness $\Delta_{fairness}$ (lower is better).

Fig. 13: CG and MG when β_{MG} is varying from 0.25 to 4 and when both applications have six cores.

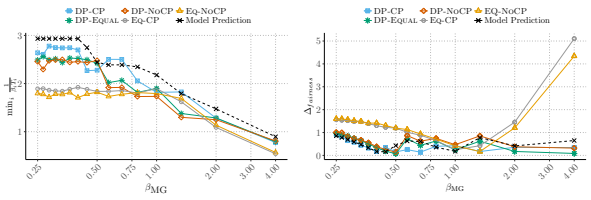


Fig. 14: Minimum throughput and $\Delta_{fairness}$ for 2CG+MG.

the proposed dynamic programming algorithm is the method that best minimizes the fairness $\Delta_{fairness}$, even though the cache partitioning with DP-CP and DP-EQUAL does not bring any clear advantage

in this scenario. This is mainly due to the fact that the application with the varying weight is a compute-intensive application, co-scheduled with two memory-intensive applications. According to our experiments, when compute-intensive applications are outnumbered by memory-intensive applications, the cache partitioning is often less efficient.

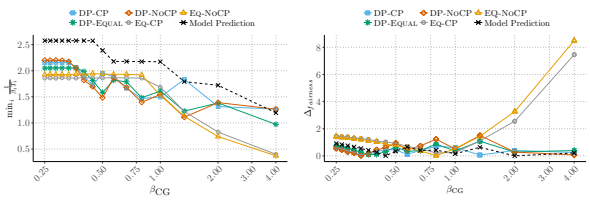


Fig. 15: Minimum throughput and $\Delta_{fairness}$ for 2MG+CG.

Figures 16, 17 and 18 also present the minimal throughput obtained when we co-schedule, respectively, 2MG+BT, 2MG+LU and 2MG+SP. 2MG co-scheduled with BT, LU or SP lead to the same behavior for the minimum throughput and the fairness $\Delta_{fairness}$, the variants based on our dynamic algorithm DP-CP, DP-Equal and DP-NoCP perform better than Eq-CP and Eq-NoCP. The fairness $\Delta_{fairness}$, for the three cases, is very large. The reason behind the large values of $\Delta_{fairness}$ is that MG is very small compared to LU, BT and SP.

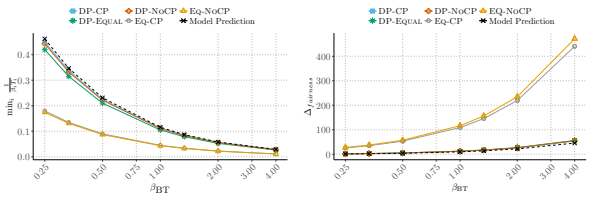


Fig. 16: Minimum throughput and $\Delta_{fairness}$ for 2MG+BT.

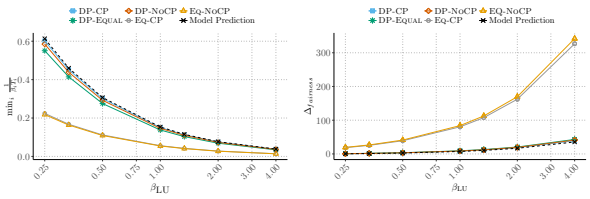


Fig. 17: Minimum throughput and $\Delta_{fairness}$ for 2MG+LU.

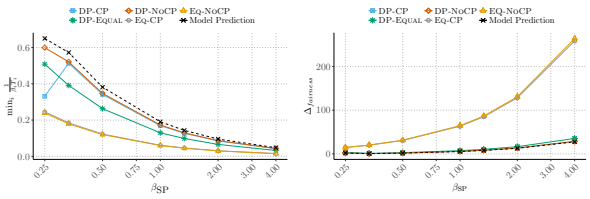


Fig. 18: Minimum throughput and $\Delta_{fairness}$ for 2MG+SP.

CG+MG+FT. Figure 19 shows the minimum throughput obtained when co-scheduling the three different applications, while varying only the weight β_{FT} of FT. We observe that the performance of the three DP-based algorithms is close to the performance obtained with the equal-resource assignment for β_{FT} smaller than 0.5, but for the other cases, DP-CP and all its variants outperform Eq-CP and Eq-NoCP. The fairness $\Delta_{fairness}$ leads to the same conclusion: DP-CP, DP-NoCP and DP-Equal are much closer to zero than Eq-CP and Eq-NoCP, especially when β_{FT} is larger than 0.5.

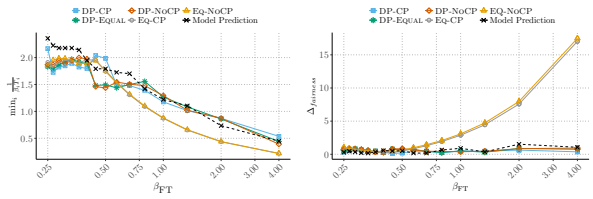


Fig. 19: Minimum throughput and $\Delta_{fairness}$ for CG, MG and FT.

Next, Figure 20 is the counterpart of Figure 19 when varying only the weight β_{MG} of MG. The results obtained by the DP-based algorithms are very good with an average gain around 50% over the Eq-CP variants, especially when β_{MG} is below 1. We note that the cache partitioning does not take advantage of this scenario, DP-CP shows degraded performance compared to DP-NoCP. For the fairness $\Delta_{fairness}$, the method that performs best is DP-CP, close to DP-NoCP and DP-Equal though.

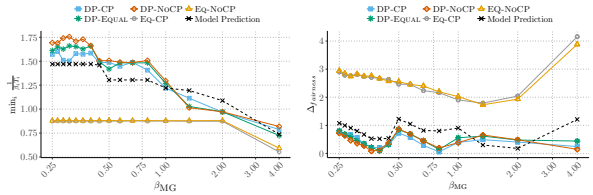


Fig. 20: Minimum throughput and $\Delta_{fairness}$ for CG, FT and MG.

Finally, Figure 21 is the counterpart of Figures 19 and 20 when varying only β_{CG}. The behavior of all DP-CP variants is interesting: for $0.25 \leq \beta_{\text{CG}} \leq 0.44$, the resource allocation, both for cores and cache, does not change, resulting into the decreasing of the minimum weighted throughput when β_{CG} is increasing (so $\frac{1}{\beta_{\text{CG}} T_{\text{CG}}}$, which is actually the minimum here, is decreasing). At $\beta_{\text{CG}} = 0.5$, the allocation of resources changes for DP-CP variants (more and more resources are allocated to CG, in order to fit the increasing requirement). We observe that DP-CP, DP-Equal and DP-NoCP logically outperform Eq-CP and Eq-NoCP to maximize the minimum weighted throughput among the co-scheduled applications. However, the cache partitioning does not help in this scenario, mainly because we vary the weight of the only compute-intensive application. In terms of fairness $\Delta_{fairness}$, obviously DP-CP, DP-Equal and DP-NoCP perform better than Eq-CP and Eq-NoCP. Among DP-CP, DP-Equal and DP-NoCP, we see that the cache partitioning version is the best method to minimize the fairness $\Delta_{fairness}$.

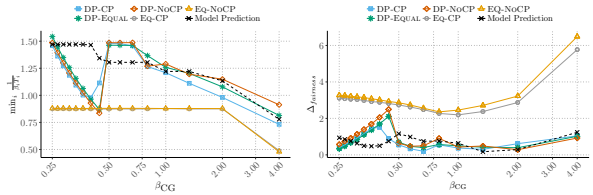


Fig. 21: Minimum throughput and $\Delta_{fairness}$ for MG, FT and CG.

Summary. Overall, we showed that we can obtain significant gains using cache partitioning (CP) when co-scheduling three applications, but it is not always the case. The difficulty of obtaining some gain with CP increases with the number of applications involved. The first reason lies in the cache size, often too small to be efficiently partitioned between the applications. The second reason is related to the behavior of the co-scheduled applications. The results show that co-scheduling one or two compute-intensive applications, such as CG, plus one memory-intensive application, such as MG, is a good way to achieve significant improvements with CP. CG is a compute-intensive kernel that performs a lot of irregular memory accesses, while MG is a memory-intensive kernel, hence if we co-schedule one CG and one MG, MG will frequently evict cache lines belonging to CG, which will slow down its execution.

7.5 Scalability and accuracy results

Scalability results. In order to study the scalability of the results, we have plotted on Figure 22 the total number of iterations done by DP-CP and DP-NoCP when varying the maximal fraction of cache used (from 10 to 20, i.e., from 50% to 100%). The only case where cache partitioning is useful is for the combination of CG with MG when a large part of the cache is available (at least 85%). This is in line with previous results, i.e., it is beneficial to schedule compute-intensive applications with a memory-intensive application. This figure leads us to think that future architectures with larger caches may benefit more from cache partitioning for these kinds of applications.

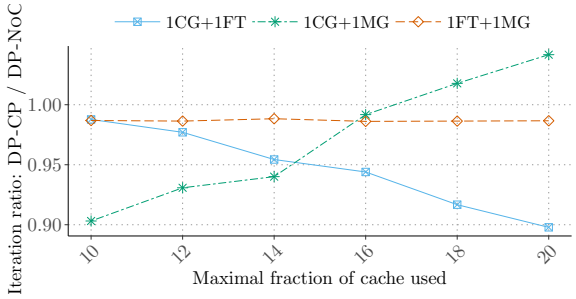


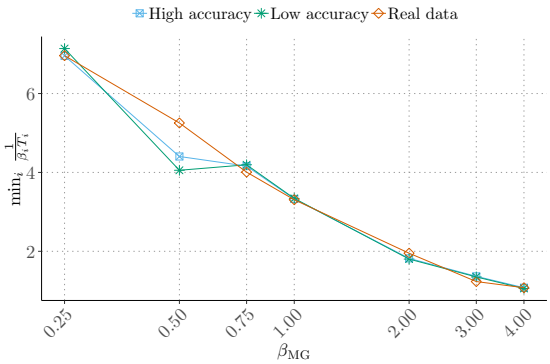
Fig. 22: Iteration ratio when using cache partitioning compared to the solution without cache partitioning.

Accuracy results. In order to study the impact of model accuracy on the performance, we have plotted on Figure 23 the minimum throughput done by DP-CP when varying the weight of the second application for three different models. We instantiate the DP-CP algorithm with two different interpolations: the first one uses $P \times X$ points where P is the total number of processors and X the total number of cache slices, denoted *High accuracy* on Figure 23. The second one, denoted *Low accuracy*, is an interpolation that only uses $P + X$ (29) values instead of $P \times X$ (220): to obtain a_i, b_i and s_i we only consider cache slices from 15% to 85% with 1 processor, and from 1 to 14 processors with 100% of the cache slices. Finally, we also compared the performance obtained by running DP-CP with the real experimental values $T_i^{real}(p_i, x_i)$, depicted as *Real data* on Figure 23. We observe that decreasing the accuracy of the model by interpolating much less points still leads to good results. This leads us to think that our approach is robust enough to reduce the amount of data needed to feed the model and thus, gain in scalability.

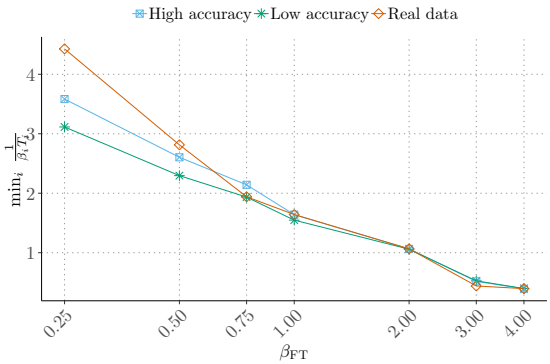
8 Conclusion

We have investigated the problem of co-scheduling iterative HPC applications, using the CAT technology provided by Intel to partition the cache. We have proposed a model for the execution time of each application, given a number of cores and a fraction of cache, and we have shown how to instantiate the model on applications coming from the NAS benchmarks. The model turns out to be accurate, as shown in the experiments where we compare the execution time predicted by the model to the real execution time. Several scheduling strategies have been designed, with the goal to maximize the minimum weighted throughput of each application. In particular, we have introduced an optimal strategy for the model, based upon a dynamic programming algorithm. The results demonstrate that in practice, the optimal strategy often leads to better results than a naive strategy sharing equally the resources between applications. Also, we have determined which combinations of applications benefit most from cache partitioning, and demonstrated the usefulness of cache partitioning.

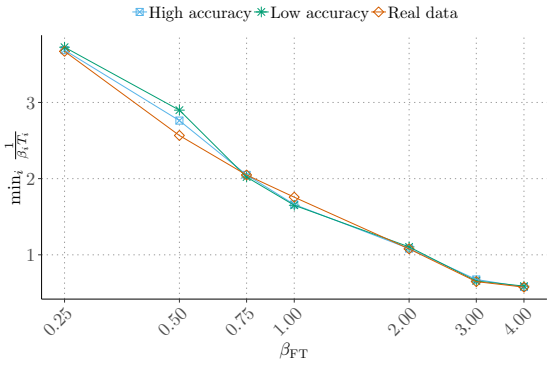
Future work will be devoted to pursuing this experimental study. We hope to get access to platforms with larger shared caches, so that we could scale up the experiments and confirm the usefulness of cache partitioning techniques.



(a) CG+MG.



(b) CG+FT.



(c) MG+FT.

Fig. 23: Minimum throughput when using DP-CP with three input models.

References

[ABD+18] Guillaume Aupy, Anne Benoit, Sicheng Dai, Loïc Pottier, Padma Raghavan, Yves Robert, and Manu Shantharam. Co-scheduling Amdahl applications on cache-partitioned systems. *The Int. Journal of High Performance Computing Applications*, 32(1):123–138, 2018.

[Amd67] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, 1967.

[B+91] D. H. Bailey et al. The NAS Parallel Benchmarks - Summary and Preliminary Results. In *Proc. of the 1991 ACM/IEEE Conf. on Supercomputing*, 1991.

[BAA+16] Andrew C Bauer, Hasan Abbasi, James Ahrens, Hank Childs, Berk Geveci, Scott Klasky, Kenneth Moreland, Patrick O'Leary, Venkatram Vishwanath, Brad Whitlock, et al. In situ

methods, infrastructures, and applications on high performance computing platforms. In *Computer Graphics Forum*, volume 35, pages 577–597. Wiley Online Library, 2016.

[BCSM08] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *4th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 101–110. IEEE Computer Society, 2008.

[BDG+00] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. A portable programming interface for performance evaluation on modern processors. *The international journal of high performance computing applications*, 14(3):189–204, 2000.

[BHP+17] Shunxing Bao, Yuankai Huo, Prasanna Parvathaneni, Andrew J Plassard, Camilo Bermudez, Yuang Yao, Ilwoo Llyu, Aniruddha Gokhale, and Bennett A Landman. A data colocation grid framework for big data medical image processing-backend design. *arXiv preprint arXiv:1712.08634*, 2017.

[Com17] PEZY Computing. Zettascaler-2.0 configurable liquid immersion cooling system, 2017.

[DR14] Matthieu Dreher and Bruno Raffin. A Flexible Framework for Asynchronous In Situ and In Transit Analytics for Scientific Simulations. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Chicago, United States, May 2014. IEEE Computer Science Press.

[Eri17] Erich Strohmaier et al. The top500 benchmark, 2017. `https://www.top500.org/`.

[GSYY09] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In *Proc. 7th ACM Int. Conf. Embedded Software*, EMSOFT '09, pages 245–254. ACM, 2009.

[HSPE08] Allan Hartstein, Vijayalakshmi Srinivasan, T Puzak, and P Emma. On the nature of cache miss behavior: Is it $\sqrt{2}$. *The Journal of Instruction-Level Parallelism*, 10:1–22, 2008.

[KCS04] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122. IEEE Computer Society, 2004.

[KSS12] Anil Krishna, Ahmad Samih, and Yan Solihin. Data sharing in multi-threaded applications and its impact on chip design. In *Int. Symp. Performance Analysis of Systems and Software (ISPASS)*, pages 125–134. IEEE, 2012.

[LCG+16] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Improving resource efficiency at scale with Heracles. *ACM Transactions on Computer Systems (TOCS)*, 34(2), 2016.

[LK14] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *9th European Conf. on Computer Systems*, 2014.

[LLD+08] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 367–378. IEEE, 2008.

[MSM+11] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *Proc. 44th IEEE/ACM Int. Sym. Microarchitecture*, MICRO-44, pages 374–385. ACM, 2011.

[MVM+15] Preeti Malakar, Venkatram Vishwanath, Todd Munson, Christopher Knight, Mark Hereld, Sven Leyffer, and Michael E Papka. Optimal scheduling of in-situ analysis for large-scale scientific simulations. In *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis, SC'15*, 2015.

[Ngu16] Khang T Nguyen. Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family, February 2016. `https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology`.

[NLS07] Kyle J Nesbit, James Laudon, and James E Smith. Virtual private caches. *ACM SIGARCH Computer Architecture News*, 35(2):57–68, 2007.

[QP06] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 423–432. IEEE, 2006.

[RKB+09] Brian M Rogers, Anil Krishna, Gordon B Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: challenges in and avenues for CMP scaling. *ACM SIGARCH Computer Architecture News*, 37(3):371–382, 2009.

[S+15] Christopher Sewell et al. Large-scale compute-intensive analysis via a combined in-situ and co-scheduling workflow approach. In *Proc. of the Int. Conf. for High Perf. Computing, Networking, Storage and Analysis, SC'15*, 2015.

[TASS07] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing shared l2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 26–33. Citeseer, 2007.

[TDF90] George Taylor, Peter Davies, and Michael Farmwald. The tlb slice-a low-cost high-speed address translation mechanism. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pages 355–363. IEEE, 1990.

[TJS09] Kai Tian, Yunlian Jiang, and Xipeng Shen. A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. In *Proc. 6th ACM Conf. Computing Frontiers*, CF '09, pages 41–50. ACM, 2009.

[ZBF10] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *ACM Sigplan Notices*, 45(3):129–142, 2010.

[ZLMT14] Yunqi Zhang, Michael A Laurenzano, Jason Mars, and Lingjia Tang. Smite: Precise QOS prediction on real-system SMT processors to improve utilization in warehouse scale computers. In *Proc. of the 47th Int. Symp. on Microarchitecture*, pages 406–418, 2014.

[ZSB+12] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys (CSUR)*, 45(1):4, 2012.