Co-scheduling HPC workloads on cache-partitioned CMP platforms

Guillaume Aupy*, Anne Benoit†§, Brice Goglin*, Loïc Pottier†, Yves Robert†‡
*Inria, Labri, Univ. Bordeaux, CNRS, Bordeaux-INP, France
†Laboratoire LIP, École Normale Supérieure de Lyon, France
‡University of Tennessee, Knoxville TN, USA
§Georgia Institute of Technology, Atlanta, GA, USA

***Abstract*—Co-scheduling techniques are used to improve the throughput of applications on chip multiprocessors (CMP), but sharing resources often generates critical interferences. 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 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.**

***Index Terms*—Co-scheduling; cache-partitioning; multiprocessor chips; cache allocation technology (CAT);**

I. Introduction

Co-scheduling applications on a chip multiprocessor (CMP) has received a lot of attention recently [1], [2]. 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 [3]: it uses PEZY-SC2, a 2048-core processor chip sharing a 40MB last level cache (LLC) [4]: 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. Which fraction of computing resources 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.

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 [5], [6] 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 [7], [8], [9] (see Section II on related work for more details).

Intel recently introduced a new hardware feature for cache partitioning called *Cache Allocation Technology* (CAT) [10]. 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. [2] 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 [11]. 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 [12]. In both cases, several applications (various computational kernels for data analysis) have to run concurrently to analyze the data in parallel of 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 [13]. 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 data 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 II

provides an overview of related work. Section III details the main framework and all application/platform parameters, as well as the optimization problem. Section IV presents five co-scheduling strategies, including a dynamic programming approach that provides an optimal resource assignment (according to the model). Section V describes the real cache partitioned platform used to perform the experiments. Section VI assesses the accuracy of the model. Section VII reports extensive experiments. Finally, Section VIII summarizes our main contributions and discusses directions for future work.

II. Related work

Recent multi-core processors show dozens of cores and a shared cache always larger. In this context, co-scheduling has been extensively studied [1], [2]. 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 [14]. 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 analyzes are co-located in the same node, while in the *in-transit* approach, the data analyzes are outsourced onto dedicated nodes [12]. Several studies have shown that large-scale simulations with *in-situ* could benefit from co-scheduling approaches [11], [15]. 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 sharing the CMP?

Shared resources include cache, memory, I/O channels and network links, but among potential degradation factors, cache accesses are prominent [16]. 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 [17], [18], [19] and software techniques [20], [21], [22], [23]. 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. Tam et al. [21], 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 [22], the cache partitioning is refined and adjusted periodically at runtime, with the objective to maximize platform efficiency.

Modeling application interference is a challenging task, Hartstein et al.. [24] 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 [25] using this Power law, we focused 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) in order to minimize the total execution time. The problem was shown to be NP-complete and heuristics were provided. We used simulations to assess the performance of these heuristics, because at that time no cache partitioning technologies were available. Intel recently released a new software technique to internally partition the last level cache (LLC), called the *Cache Allocation Technology* (CAT) [10], [2]. In this paper, we use CAT to experiment with a real cache partitioned platform. The main differences with our previous work are as follows. In this work: (i) we use an application model that renders the problem tractable and is still sufficiently precise; (ii) we target a real-life optimization objective that corresponds to weighted throughput in steady-state processing; (iii) we provide an optimal polynomial algorithm for this objective function; (iv) finally, we perform actual implementations and experiments to verify our model and algorithms. To the best of our knowledge, this work is the first co-scheduling study for a cache partitioned system (using CAT) with HPC workloads.

III. Model and optimization problem

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, which 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 $T_i^{real}(p_i, x_i)$. On a given platform, all these values can be measured, and we aim at providing a model that characterizes these values. In the model, we use the following formula:

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

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, i.e., $t_i(p_i)$ and $h_i(x_i)$ are non-increasing functions. In this formula, we assume that the slowdown incurred by cache misses does not

depend on the number of cores assigned to the application. While this assumption may not be true in practice, we will discuss the model accuracy in Section VI, where we measure cache misses and refine the model.

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

A. Computations $t_i(p_i)$

We assume that all applications obey Amdahl's law [26]: $t_i(p_i) = s_i T_i^{seq} + (1 - s_i)\frac{T_i^{seq}}{p_i}$, 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.

B. 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 [24], [27], [28]. The Power Law of cache misses states that for a cache of size C_{act}, the cache miss ratio r can be expressed as

$$r = r_0 \left(\frac{C_0}{C_{act}} \right)^{\alpha}, \tag{2}$$

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:

$$h_i(x_i) = a_i + \frac{b_i}{\sqrt{x_i}}, \tag{3}$$

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

Overall, when assigning p_i cores and a fraction x_i of the cache, and letting $c_i = 1 + a_i$, an application A_i executes one iteration in time:

$$T_i(p_i, x_i) = t_i(p_i)\left(c_i + \frac{b_i}{\sqrt{x_i}} \right). \tag{4}$$

C. Optimization problem

As stated in Section I, 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 [13]. We let β_i denote the weight of application A_i for $1 \leq i \leq m$. 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 $\frac{1}{\beta_i T_i(p_i,x_i)}$, and 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. Note that letting $\beta_i = 1$ leads 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*

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

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

IV. 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.

A. 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 (4).

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 III.

B. 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.

C. 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.

V. EXPERIMENTAL SETUP

In this section, we first describe the platform and the benchmark applications in Section V-A. Then in Section V-B, we explain in details the *Cache Allocation Technology* CAT.

A. 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 vanilla 4.11.0 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 VI) or 5 (in Section VII) 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 [29].

For validation and performance evaluation, we use HPC workloads from class A NAS benchmarks [30]. Due to lack of space, we focus on three applications CG, MG and FT, which exhibit the most interesting behaviors. (see Table I). Three additional NAS benchmarks are considered in the extended version [31].

App	Description
CG	Uses conjugate gradients method to solve a large sparse symmetric positive definite system of linear equations
MG	Performs a multi-grid solve on a sequence of meshes
FT	Performs discrete 3D fast Fourier Transform

Table I: Description of the NAS parallel benchmarks.

B. Cache Allocation Technology

The Cache Allocation Technology (CAT) [10] is part of a larger set of Intel technologies that are called the Intel Resource Director Technology (RDT) support 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. Each COS 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. The set of slices available to a class is a capacity bit-mask (CBM) of length X. 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).

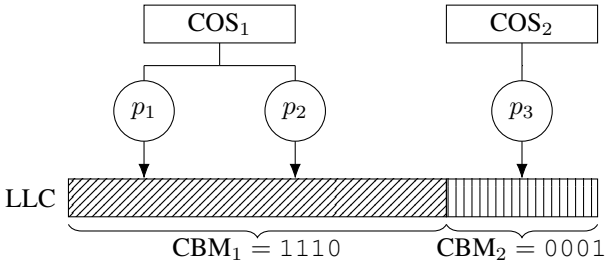

Figure 1: CAT example with 2 classes of service, 3 cores and a 4-bit capacity mask (CBM). First COS has 2 cores and 75% of the LLC, the second class of service has the remaining resources.

VI. Accuracy of the model

In this section, we assess the precision of the model introduced in Section III. First, we detail the experimental protocol and explain how to obtain the model parameters for each application in Section VI-A. Then in Section VI-B, we study the behavior of cache misses on the platform described in Section V-A. Finally, we study in Section VI-C the accuracy of the model by comparing the expected execution time from Equation (4) to the measured one T^{real}.

A. 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 (4): s_i (sequential fraction), and a_i (or equivalently $c_i = a_i + 1$) and b_i (cache slowdown). To this purpose, we monitor each application with PAPI [29] 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 (4).

We observe 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 like MG and FT. The results for each application are displayed in Table II (more detailed results are available in [31]).

App$_i$	a_i	b_i	s_i
CG	-0.0379	0.0474	0
FT	0.0092	0.0129	0.016
MG	0.0460	0.0073	0.065

Table II: Parameters s_i, a_i and b_i obtained by interpolation from the data produced by measurements.

B. Cache miss behavior

Figure 2 shows the evolution of cache miss ratios for the three 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. Hence, most of the time, the assumption taken in the model that the slowdown incurred by cache misses does not depend on the number of cores is acceptable, even though there might be some differences, in particular for the MG application.

C. Accuracy of the execution time

Finally, we aim at verifying the accuracy of the execution time predicted by the model. Figure 3 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 3, the number of cores varies from 1 to 14 while the cache fraction is fixed at $x = 3$ (or 15%).

Figure 4 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 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

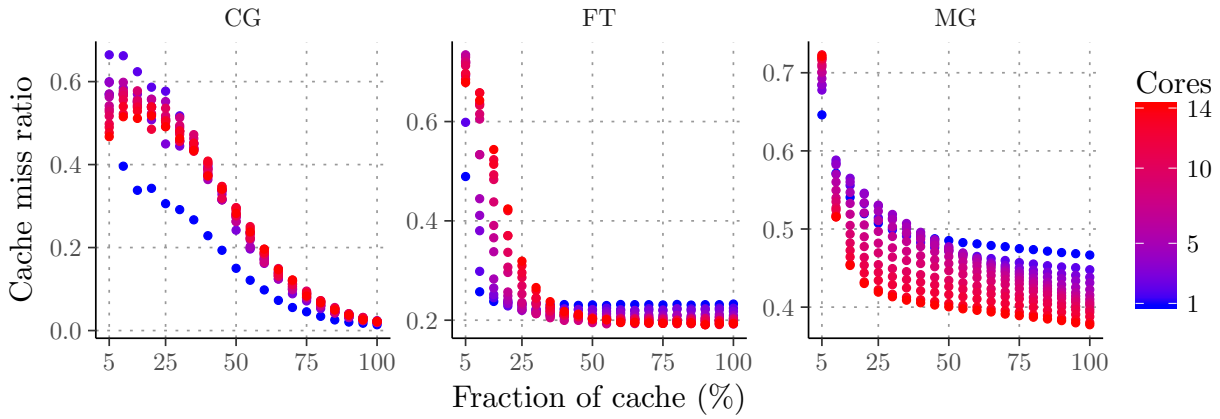

Figure 2: Evolution of cache miss ratio with x_i and p_i.

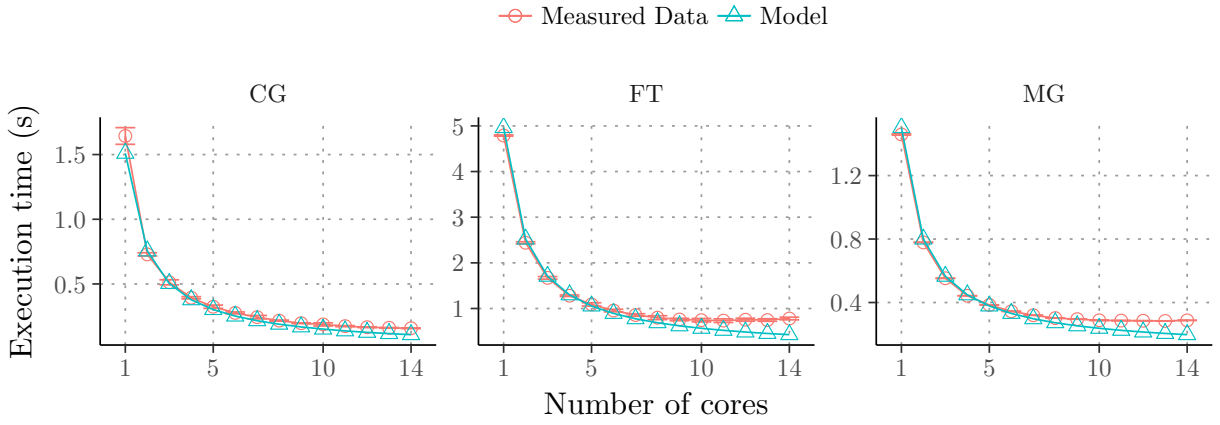

Figure 3: Comparison between predicted execution time by the model and measured execution time, when varying the number of cores up to 14 and with a cache fraction set to 15%.

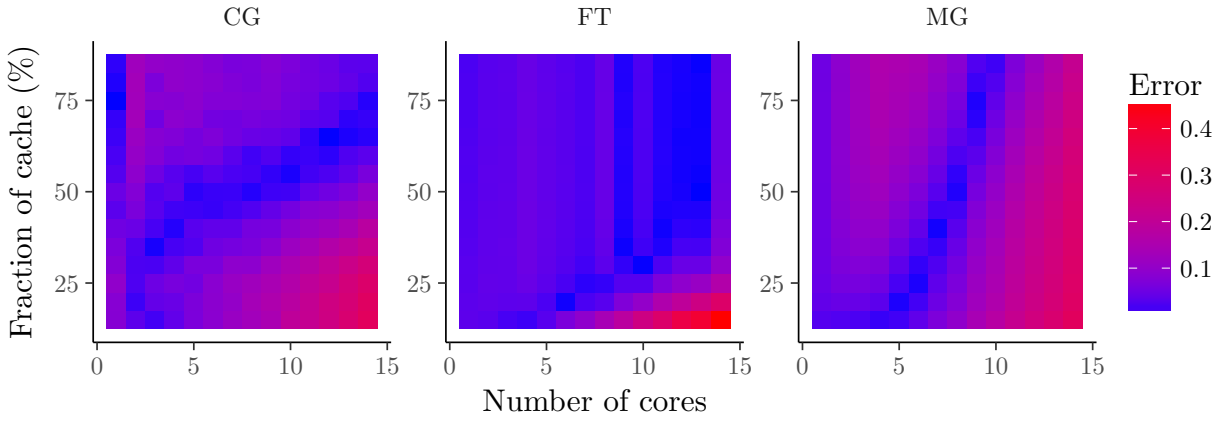

Figure 4: Heat-map of the relative error.

and high number of cores. This is due to a specific behavior of FT: its execution time tends to become constant after a certain core threshold (see Figure 3), while the model expects a strictly decreasing execution time. This constant plateau is not due to Amdahl's law (FT is 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 mispredictions when the number of cores increases, is that the model assumes that the number of cores does not impact LLC cache misses, which is not always true in practice, as seen in Figure 2.

D. Summary

We have checked that the model is relatively accurate, even though it takes some simplifying assumptions that are not completely true in practice (cache misses independent of the number of cores). The next section assesses the performance of the scheduling strategies, in particular when using the model.

VII. Results

To assess the performance of the scheduling strategies of Section IV 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.

A. Experimental protocol

The platform and the applications used for all the experiments are described in Section V. 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 $T = 3$ minutes, which is large enough to ensure that each application reaches the steady state with enough iterations. 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. Similar results were obtained when co-scheduling applications on all 14 cores.

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 by 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, $\Delta_{fairness}$:

$$\Delta_{fairness} = \sum_{i \neq j} \left| \frac{\beta_i T_i}{\beta_j T_j} - 1 \right|. \quad (5)$$

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.

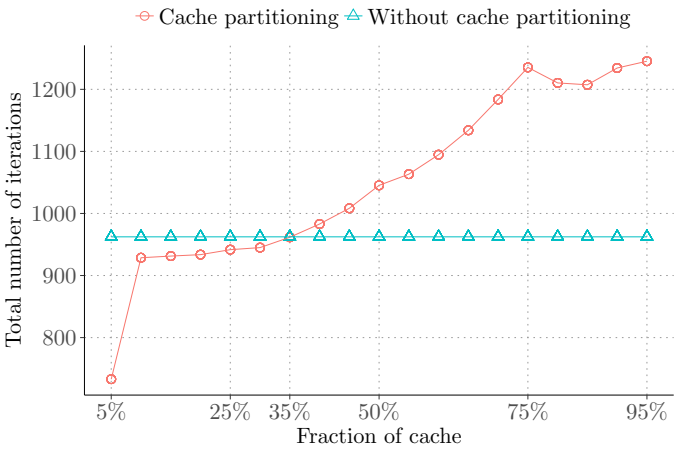

Figure 5: CG and MG (six cores each).

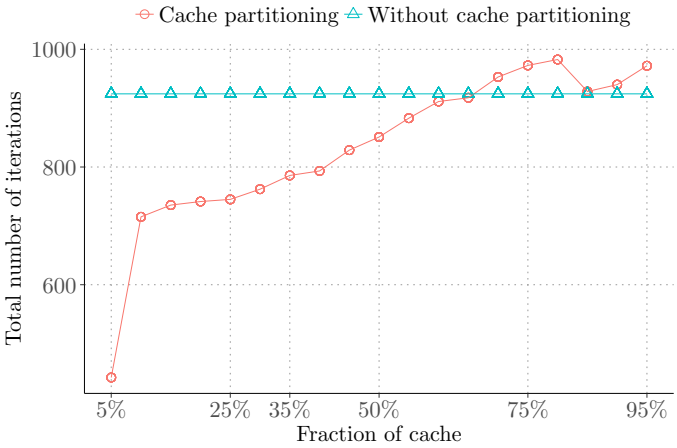

Figure 6: CG and FT (six cores each).

B. 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, conversely, 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 cache partitioning enabled or not.

Figure 5 shows the impact of CP for CG+MG: we 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 beyond 35%, while the performance of CG greatly depends on the cache size (for more details on application behaviors, see Figure 2). Without a

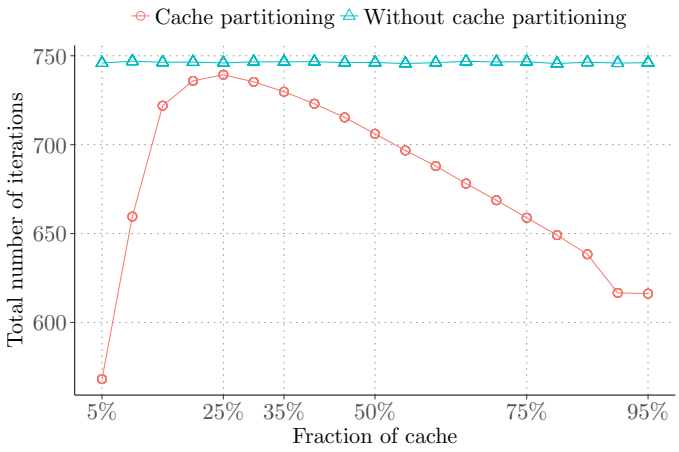

Figure 7: FT and MG (six cores each).

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.

For the other combinations CG+FT or FT+MG (see Figures 6 and 7), we see little improvement, since FT is more communication intensive (all-to-all communication) than strictly memory intensive, hence the gain obtained by cache partitioning is less important than for CG+MG. The worst case for cache partitioning is when combining FT with MG, since both applications are memory and communication intensive, and hence none of them needs a strict cache partitioning, since their use of the cache varies during iterations.

Overall, 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 cache partitioning.

C. 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 IV. 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 every β_i iterations of A_i, we aim at performing β_j iterations of A_j. To avoid some side 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 and we slightly modify the dynamic algorithm accordingly), while the minimum number of cores per application is set to one.

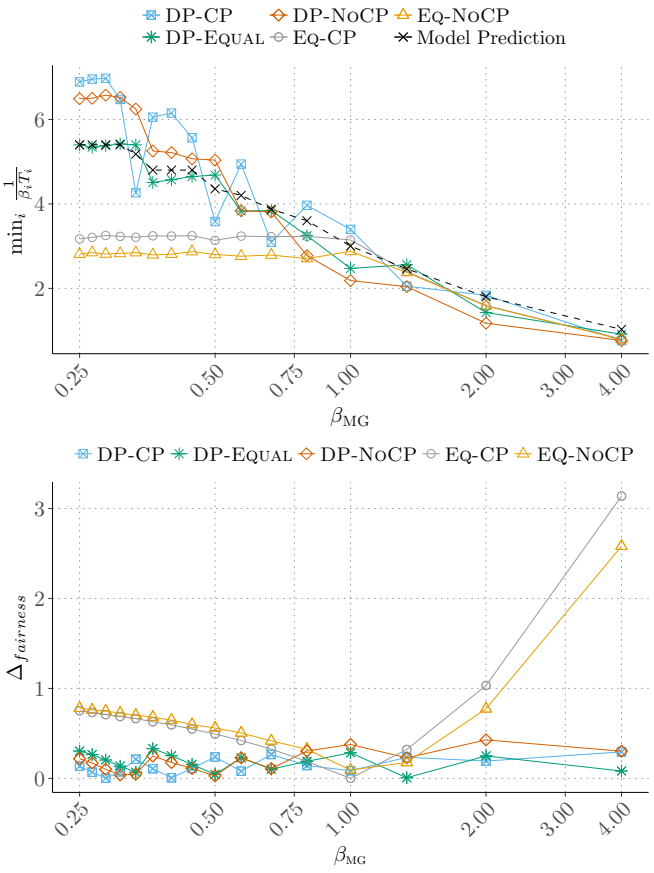

Figure 8: Minimum throughput and $\Delta_{fairness}$ for CG and MG.

We focus again on CG+MG, since this combination turned out to be the most interesting in terms of cache partitioning. On Figure 8 (top), we see what is the minimum throughput achieved by each method for CG+MG. The weight β_{MG} of 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. On the same figure, we also depict the model prediction, as the minimum throughput computed from $T_i(p_i, x_i)$ values with p_i and x_i derived from our optimal algorithm DP-CP. We observe that the model is accurate enough to fits well the performance of DP-CP obtained on our experimental platform. Figure 8 (bottom) presents $\Delta_{fairness}$, as defined in Equation (5). 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.

For the other combinations (CG+FT and MG+FT, see [31]), we observe similar results, where the DP-based algorithms almost systematically outperform Eq-CP and Eq-NoCP. Also, in most cases, the variants using cache partitioning perform better than those without cache partitioning, even

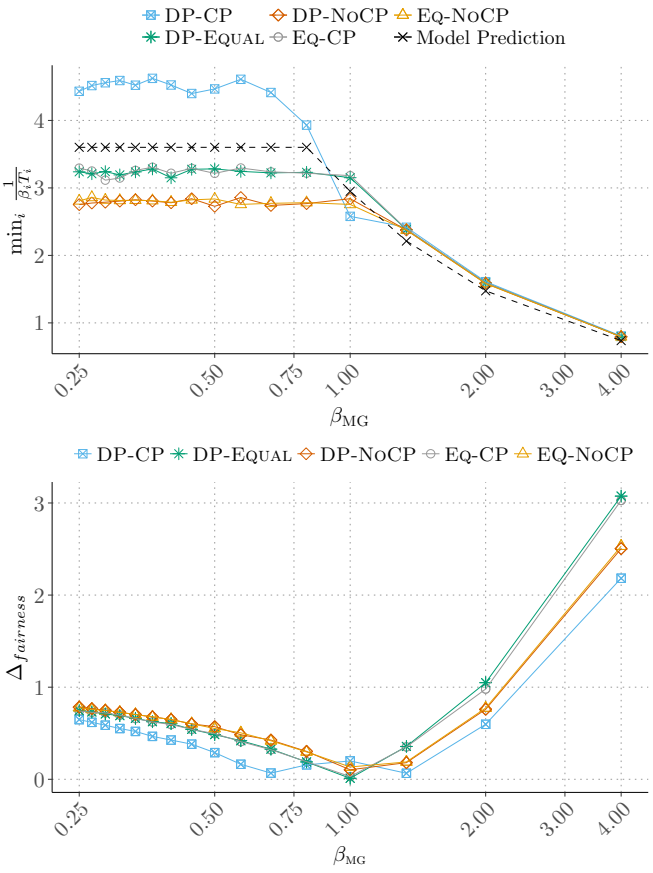

Figure 9: Minimum throughput and $\Delta_{fairness}$ for CG and MG, where both applications have six cores.

though FT benefits less from cache partitioning than when we combine CG with MG. In terms of fairness, the DP-based algorithms are always very close to zero.

We also consider a special case where all applications have the same number of cores (six in our case), so only the cache is available to increase performance, see Figure 9. In this case, DP-CP is the only method that can choose how to partition the cache, and it succeeds to obtain up to 25% improvement when β_{MG} is smaller than 1 (compute more CG than MG). Also, even though $\Delta_{fairness}$ is high for all methods in this setting (since they cannot modify the number of cores), the error of DP-CP is the smallest. We also notice that our model prediction is pretty close to the experimental results.

Summary: Overall, 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 9 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.

D. Co-scheduling results with three applications

Similarly to the case with two applications, with three applications (A_1, A_2, A_3), β_3 is ranging from 0.25 to 4, while $\beta_1 = \beta_2 = 1$. We present only two examples of co-schedules; more combinations are available in [31], with no significantly different conclusions. Experiments with different values of β_1 and β_2 also provide similar results.

Figure 10 shows the minimum throughput obtained when we co-schedule 2CG+MG, while the weight β_{MG} 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, namely DP-CP, DP-Equal and DP-NoCP, outperform Eq-CP and Eq-NoCP. In addition, cache partitioning shows a great interest here: DP-CP exhibits a gain around 15% on average over DP-NoCP and DP-Equal. The model prediction is also very accurate with three applications, even more accurate than with two applications. This difference of accuracy mainly lies in the fact that our model is more accurate when p_i and x_i are not extreme (close to the minimum or the maximum possible), which happens when the number of applications increases. The fairness criterion $\Delta_{fairness}$ is also depicted. Recall that ideally, we would like to have $\beta_i T_i = \beta_j T_j$ for all i, j (see Equation (5)). We observe that the method that is the closest to zero is DP-CP, confirming the strong interest of cache partitioning.

Figure 11 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 criterion $\Delta_{fairness}$ 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.

Summary: Overall, we showed that we can obtain important 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 evict

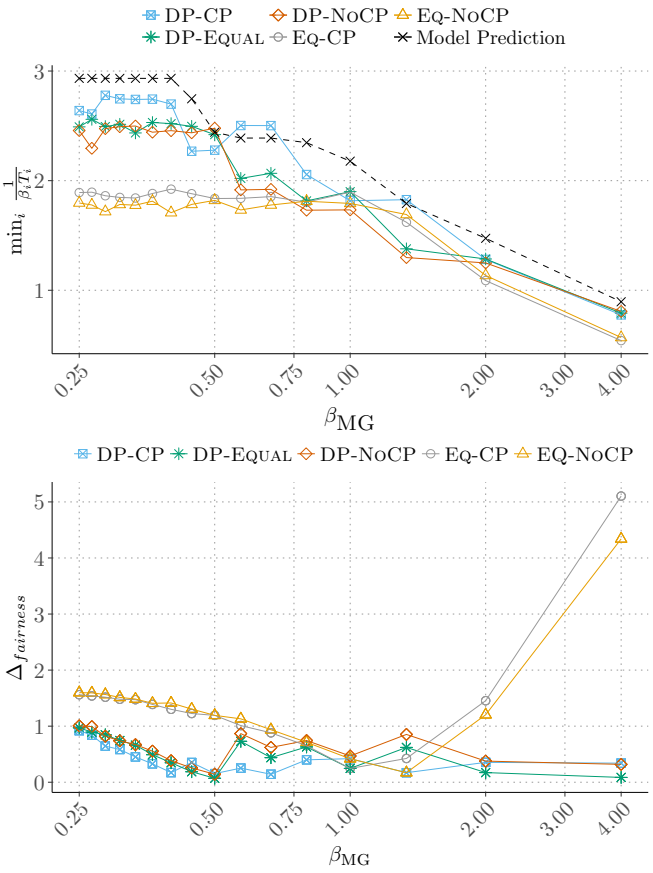

Figure 10: Minimum throughput and $\Delta_{fairness}$ for 2CG+MG.

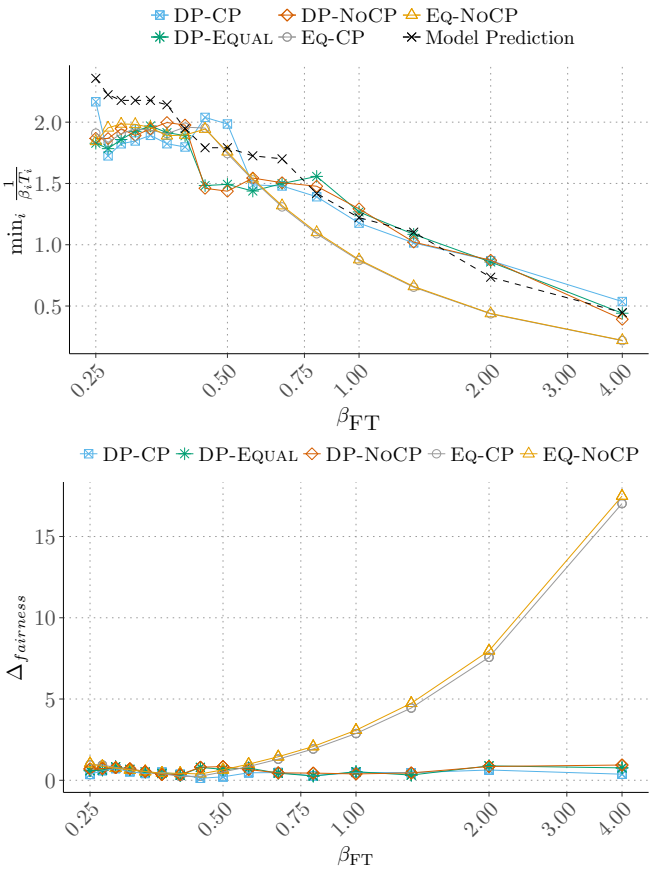

Figure 11: Minimum throughput and $\Delta_{fairness}$ for CG, MG and FT.

very often cache lines belonging to CG, which will slow down its execution.

VIII. 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 extending 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. We will also generalize the experiments to multiprocessors and see if there is a benefit in moving applications from one processor to another, in order to avoid co-locating several cache-intensive applications on the same processor. Another interesting direction would be to consider the Universal Scalability Law [32] instead of Amdahl's law, thereby generalizing the model in order to account for contentions.

REFERENCES

[1] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *Proc. 44th IEEE/ACM Int. Sym. Microarchitecture*, ser. MICRO-44. ACM, 2011, pp. 374–385.

[2] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Improving resource efficiency at scale with Heracles," *ACM Transactions on Computer Systems (TOCS)*, vol. 34, no. 2, 2016.

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

[4] P. Computing, "Zettascaler-2.0 configurable liquid immersion cooling system," 2017. [Online]. Available: http://www.exascaler.co.jp/wp-content/uploads/2017/11/zettascaler2.0_en_page.pdf

[5] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *9th European Conf. on Computer Systems*, 2014.

[6] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," *ACM Sigplan Notices*, vol. 45, no. 3, pp. 129–142, 2010.

[7] Y. Zhang, M. A. Laurenzano, J. Mars, and L. 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*, 2014, pp. 406–418.

[8] 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*. IEEE Computer Society, 2008, pp. 101–110.

[9] K. Tian, Y. Jiang, and X. Shen, "A study on optimally co-scheduling jobs of different lengths on chip multiprocessors," in *Proc. 6th ACM Conf. Computing Frontiers*, ser. CF '09. ACM, 2009, pp. 41–50.

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

[11] C. 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.

[12] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock *et al.*, "In situ methods, infrastructures, and applications on high performance computing platforms," in *Computer Graphics Forum*, vol. 35. Wiley Online Library, 2016, pp. 577–597.

[13] P. Malakar, V. Vishwanath, T. Munson, C. Knight, M. Hereld, S. Leyffer, and M. 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.

[14] M. Dreher and B. 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: IEEE Computer Science Press, May 2014. [Online]. Available: https://hal.inria.fr/hal-00941413

[15] S. Bao, Y. Huo, P. Parvathaneni, A. J. Plassard, C. Bermudez, Y. Yao, I. Llyu, A. Gokhale, and B. A. Landman, "A data colocation grid framework for big data medical image processing-backend design," *arXiv preprint arXiv:1712.08634*, 2017.

[16] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 4, 2012.

[17] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2004, pp. 111–122.

[18] M. K. Qureshi and Y. 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*. IEEE, 2006, pp. 423–432.

[19] K. J. Nesbit, J. Laudon, and J. E. Smith, "Virtual private caches," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 57–68, 2007.

[20] G. Taylor, P. Davies, and M. Farmwald, "The tlb slice-a low-cost high-speed address translation mechanism," in *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*. IEEE, 1990, pp. 355–363.

[21] D. Tam, R. Azimi, L. Soares, and M. Stumm, "Managing shared l2 caches on multicore systems in software," in *Workshop on the Interaction between Operating Systems and Computer Architecture*. Citeseer, 2007, pp. 26–33.

[22] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. 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*. IEEE, 2008, pp. 367–378.

[23] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *Proc. 7th ACM Int. Conf. Embedded Software*, ser. EMSOFT '09. ACM, 2009, pp. 245–254.

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

[25] G. Aupy, A. Benoit, S. Dai, L. Pottier, P. Raghavan, Y. Robert, and M. Shantharam, "Co-scheduling Amdahl applications on cache-partitioned systems," *The Int. Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 123–138, 2018.

[26] G. Amdahl, "The validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS Conference Proceedings*, 1967, pp. 483–485.

[27] A. Krishna, A. Samih, and Y. Solihin, "Data sharing in multi-threaded applications and its impact on chip design," in *Int. Symp. Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2012, pp. 125–134.

[28] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: challenges in and avenues for CMP scaling," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 371–382, 2009.

[29] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *The international journal of high performance computing applications*, vol. 14, no. 3, pp. 189–204, 2000.

[30] D. H. Bailey *et al.*, "The NAS Parallel Benchmarks - Summary and Preliminary Results," in *Proc. of the 1991 ACM/IEEE Conf. on Supercomputing*, 1991. [Online]. Available: http://doi.acm.org/10.1145/125826.125925

[31] G. Aupy, A. Benoit, B. Goglin, L. Pottier, and Y. Robert, "Co-scheduling HPC workloads on cache-partitioned CMP platforms," INRIA, Res. rep. RR-9154, 2018.

[32] N. J. Gunther, *Guerrilla capacity planning - a tactical approach to planning for highly scalable applications and services*. Springer, 2007.