Co-scheduling algorithms for cache-partitioned systems

Guillaume Aupy, Anne Benoit, Loïc Pottier, Padma Raghavan, Yves Robert, Manu Shantharam

RESEARCH REPORT
N° 8965
October 2016
Project-Team ROMA

ISRN INRIA/RR--8965--FR+ENG
ISSN 0249-6399

Co-scheduling algorithms for cache-partitioned systems

Guillaume Aupy[*], Anne Benoit[†], Loïc Pottier[†], Padma Raghavan[*], Yves Robert[†‡], Manu Shantharam[§]

Project-Team ROMA

Research Report n° 8965 — October 2016 — 27 pages

Abstract: Cache-partitioned architectures allow subsections of the shared last-level cache (LLC) to be exclusively reserved for some applications. This technique dramatically limits interactions between applications that are concurrently executing on a multi-core machine. Consider n applications that execute concurrently, with the objective to minimize the makespan, defined as the maximum completion time of the n applications. Key scheduling questions are: (i) which proportion of cache and (ii) how many processors should be given to each application? Here, we assign rational numbers of processors to each application, since they can be shared across applications through multi-threading. In this paper, we provide answers to (i) and (ii) for perfectly parallel applications. Even though the problem is shown to be NP-complete, we give key elements to determine the subset of applications that should share the LLC (while remaining ones only use their smaller private cache). Building upon these results, we design efficient heuristics for general applications. Extensive simulations demonstrate the usefulness of co-scheduling when our efficient cache partitioning strategies are deployed.

Key-words: Co-scheduling; cache partitioning; complexity results.

[*] Vanderbilt University, Nashville TN, USA
[†] Laboratoire LIP, École Normale Supérieure de Lyon & Inria, France
[‡] University of Tennessee, Knoxville TN, USA
[§] San Diego Supercomputer Center, San Diego CA, USA

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Algorithmes d'ordonnancement concurrent pour systèmes à partitionnement de cache

Résumé : Les architectures à partitionnement de cache permettent d'allouer des portions du dernier niveau de cache (LLC) exclusivement réservées à certaines applications. Cette technique permet de réduire drastiquement les interactions entre applications qui sont exécutées simultanément sur un machine multi-cœurs. Considérons n applications exécutées simultanément avec l'objectif de minimiser le *makespan*, défini comme le maximum des temps de complétions parmi les n applications. Les problèmes d'ordonnancement sont les suivants: (i) quelle proportion de cache et (ii) combien de processors doivent être alloués à chaque application. Ici, nous assignons des nombres de processeurs rationnels pour chaque application, pour qu'ils puissent être partagés parmi les applications grâce au *multi-threading*. Dans ce travail, nous fournissons des réponses aux questions (i) et (ii) pour des applications parfaitement parallèles. Malgré cela, le problème est prouvé être NP-complet, et nous donnons des éléments clés pour déterminer le sous-ensemble des applications qui doivent partager le dernier niveau de cache (tandis que les autres utilisent seulement leur petit cache privé). Basé sur ces résultats, nous développons des heuristiques efficaces pour des profils d'applications généraux. Un ensemble complet de simulations démontre l'utilité de l'ordonnancement concurrent quand les techniques de partitionnement de cache sont mises en place.

Mots-clés : Ordonnancement concurrent; partitionnement de cache; résultats de complexité.

1 Introduction

At scale, the I/O movements of HPC applications are expected to be one of the most critical problems [1]. Observations on the Intrepid machine at Argonne National Laboratory (ANL) show that I/O transfers can be slowed down up to 70% due to congestion [9]. When ANL upgraded its house supercomputer from Intrepid (Peak perf: 0.56 PFlops; peak I/O throughput: 88 GB/s) to Mira (Peak perf: 10 PFlops; peak I/O throughput: 240 GB/s), the net result for an application whose I/O throughput scales linearly (or worse) with performance was a downgrade from 160 GB/PFlop to 24 GB/PFlop!

To cope with such an imbalance (which is not expected to reduce on future platforms), a possible approach is to develop *in situ* co-scheduling analysis and data preprocessing on dedicated nodes [1]. This scheme applies to data-intensive periodic workflows where data is generated by the main simulation, and parallel processes are run to process this data with the constraints that output results should be sent to disk storage before newly generated data arrives for processing. These solutions are starting to be implemented for HPC applications. Sewell et al. [26] explain that in the case of the HACC application (a cosmological code), petabytes of data are created to be analyzed later. The analysis is done by multiple independent processes. The idea of their work is to minimize the amount of data copied to I/O filesystem, by performing the analysis at the same time as HACC is running (what they call *in situ*). The main constraint is that these processes are data-intensive and are handled by a dedicated machine. Also, the execution of these processes should be done efficiently enough so that they finish before the next batch of data arrives, hence resulting in a pipelined approach. All these frameworks motivate the design of efficient co-scheduling strategies.

One main issue of co-scheduling is to evaluate co-run degradations due to cache sharing [30]. Many studies have shown that interferences on the shared last-level cache (LLC) can be detrimental to co-scheduled applications [19]. Previous solutions consisted in preventing co-schedule of possibly interfering workloads, or terminating low importance applications [28]. Lo et al. [20] recently showed experimentally that important gains could be reached by co-scheduling applications with strict cache partitioning enabled. Cache partitioning, the technique at the core of this work, consists in reserving exclusivity of subsections of the LLC of a chip multi-processor (CMP), to some of the applications running on this CMP. This functionality was recently introduced by Intel under the name *Cache Allocation Technology* [14]. With the advent of large shared memory multi-core machines (e.g., Sunway TaihuLight, the current #1 supercomputer uses 256-cores processor chips with a shared memory of 32GB [7]), the design of algorithms that co-schedule applications efficiently and decide how to partition the shared memory (seen as the cache here), is becoming critical.

In this work, we study the following problem: given a set of parallel applications, a multi-core processor with a shared last-level cache LLC, how can we best partition the LLC to minimize the total execution time (or *makespan*), i.e., the moment when the last application finishes its computation. For each application, we assume that we know the number of compute operations to perform, and the miss rate on a fixed size cache. For the multi-core processor, we know its LLC size, the cost for a cache miss, the cost for a cache hit, the size of the cache and total number of processors. We assume that these processors can be shared by two applications through multi-threading [16], hence we assign fractional number of processors to each application. In addition to being very natural in practice, this relaxation of the problem avoids artefacts due to rounding when assigning only integer numbers of processors, and allows us to study the intrinsic complexity of co-scheduling with cache partitioning. Equipped with all these applications and platform parameters, recent work [12, 25, 16] shows how to model the impact of cache misses and to accurately predict the execution time of an application. In this context, we make the

following main contributions:
• We show that the co-scheduling problem is NP-complete, even when applications are perfectly parallel, i.e., their speed-up scales up linearly with the number of processors.
• In the case of perfectly parallel applications, we show several results that characterize optimal solutions. We show that the co-scheduling cache-partitioning problem reduces to deciding which subset of applications will share the LLC; when this subset is known, we show how to determine the optimal cache fractions and number of processors.
• While these results only hold for perfectly parallel applications, they guide the design of heuristics for general applications. We show through extensive simulations that our heuristics greatly improve the performance of cache-partitioning algorithms, even for parallel applications obeying Amdahl's law with a large sequential fraction, hence with a limited speedup profile.

The rest of the paper is organized as follows. Section 2 provides an overview of related work. Section 3 is devoted to formally defining the framework and all model parameters. Section 4 gives our main theoretical contributions. The heuristics are defined in Section 5, and evaluated through simulations in Section 6. Finally, Section 7 outlines our main findings and discusses directions for future work.

2 Related work

Since the advent of systems with tens of cores, co-scheduling has received considerable attention. Due to lack of space, we refer to [22, 6, 20] for a survey of many approaches to co-scheduling. The main idea is to execute several applications concurrently rather than in sequence, with the objective to increase platform throughput. Indeed, some individual applications may well not need all available cores, or some others could use all resources, but at the price of a dramatic performance loss. In particular, the latter case is encountered whenever application speedup becomes too low beyond a given processor count.

The main difficulty of co-scheduling is to decide which applications to execute concurrently, and how many cores to assign to each of them. Indeed, when executing simultaneously, any two applications will compete for shared resources, which will create interferences and decrease their throughput. Modeling application interference is a challenging task. Dynamic schedulers are used when application behavior is unknown [24, 27]. Static schedulers aim at optimizing the sharing of the resources by relying on application knowledge such as estimated workload, speed-up profile, cache behavior, etc. One widely-used approach is to build an interference graph whose vertices are applications and whose edges represent degradation factors [15, 29, 13]. This approach is interesting but hard to implement. Indeed, the interaction of two applications depends on many factors, such as their size, their core count, the memory bandwidth, etc. Obtaining the speedup profile of a single application already is difficult and requires intensive benchmarking campaigns. Obtaining the degradation profile of two applications is even more difficult and can be achieved only for regular applications. To further darken the picture, the interference graph subsumes only pairwise interactions, while a global picture of the processor and cache requirements for all applications is needed by the scheduler.

Shared resources include cache, memory, I/O channels and network links, but among potential degradation factors, cache accesses are prominent. When several applications share the cache, they are granted a fraction of cache lines as opposed to the whole cache, and their cache miss ratio increases accordingly. Multiple cache partitioning strategies have been proposed [5, 11, 4, 8]. In this paper, we focus on a static allocation of LLC cache fractions, and processor numbers, to concurrent applications as a function of several parameters (cache-miss ratio, access frequency, operation count). To the best of our knowledge, this work is the first analytical model and complexity study for this challenging problem.

3 Model

This section details all platform and application parameters, and formally states the optimization problem.

Architecture. We consider a parallel platform of p homogeneous computing elements, or *processors*, that share two storage locations:
• A small storage $\mathcal{S}_s$ with low latency, governed by a LRU replacement policy, also called *cache*;
• A large storage $\mathcal{S}_l$ with high latency, also called *memory.*
More specifically, C_s (resp. C_l) denotes the size of $\mathcal{S}_s$ (resp. $\mathcal{S}_l$), and l_s (resp. l_l) the latency of $\mathcal{S}_s$ (resp. $\mathcal{S}_l$). In this work, we assume that $C_l = +\infty$. We have the relation $l_s \ll l_l$.

In this work, we consider the cache partitioning technique [14], where one can allocate a portion of the cache to applications so that they can execute without interference from other applications.

Applications. There are n independent parallel applications to be scheduled on the parallel platform, whose speedup profiles obey Amdahl's law [2]. For an application T_i, we define several parameters:

- w_i, the number of computing operations needed for T_i;
- s_i, the sequential fraction of T_i;
- f_i, the frequency of data accesses of T_i: f_i is the number of data accesses per computing operation;
- a_i, the memory footprint of T_i.

We use these parameters to model the execution of each application as follows.

The power law of cache misses In chip multi-processors, many authors have observed that the Power Law accurately models how the cache size affects the miss rate [12, 25, 16]. Mathematically, the power law states that if m_0 is the miss rate of a workload for a baseline cache size C_0, the miss rate m for a new cache size C can be expressed as $m = m_0 \left(\frac{C_0}{C}\right)^\alpha$ where α is the sensitivity factor from the Power Law of Cache Misses [12, 25, 16] and typically ranges between 0.3 and 0.7 with an average at 0.5. Note that, by definition, a rate cannot be higher than 1, hence we extend this definition as:

$$m = \min\left(1, m_0\left(\frac{C_0}{C}\right)^\alpha\right). \tag{1}$$

This formula can be read as follows: if the cache size allocated is too small, then the execution goes as if no cache was allocated, and all accesses will be misses.

Computations and data movement We use the cost model introduced by Krishna et al. [16] to evaluate the execution cost of an application as a function of the cache fraction that it has been allocated. Specifically, for each application, we define m_0, the miss rate of application T_i with a cache of size C_0 (we can also use the miss rate of applications with a cache of another fixed size). We express the execution time of T_i as a function of p_i, the number of processors allocated to T_i, and x_i, the fraction of $\mathcal{S}_s$ allocated to T_i (recall both are rational numbers). Let $Fl_i(p_i)$ be the number of operations performed by each processor for application T_i, given that the application is executed on p_i processors. We have $Fl_i(p_i) = s_i w_i + (1 - s_i)\frac{w_i}{p_i}$ according to Amdahl's speedup profile. Finally,

$$\mathcal{E}\mathrm{xe}_i(p_i, x_i) = \begin{cases} Fl_i(p_i)\left(1 + f_i\left(l_s + l_l \cdot \min\left(1, \frac{m_0}{\left(\frac{x_i C_s}{C_0}\right)^\alpha}\right)\right)\right) \\ \qquad \text{if } x_i C_s \le a_i; \\ Fl_i(p_i)\left(1 + f_i\left(l_s + l_l \cdot \min\left(1, \frac{m_0}{\left(\frac{a_i}{C_0}\right)^\alpha}\right)\right)\right) \\ \qquad \text{otherwise.} \end{cases} \tag{2}$$

Indeed, for each operation, we pay the cost of the computing operation, plus the cost of data accesses, and by definition we have f_i accesses per operation. At each access, we pay a latency l_s, and an additional latency l_l in case of cache miss (see Equation (1)). The second case states that we cannot use a portion of cache greater than the memory footprint a_i of application T_i. This model is somewhat pessimistic: cache accesses to the same variable by two different processors are counted twice. We show in Section 6 that despite this conservative assumption (no sharing), co-scheduling can outperform classical approaches that sequentially deploy each application on the whole set of available resources.

Equation (2) calls for a few observations. For notational convenience, let $d_i = m_0\left(\frac{C_0}{C_s}\right)^\alpha$:

- It is useless to give a fraction of cache larger than $\frac{a_i}{C_s}$ to application T_i;
- Because of the minimum $\min\left(1, \frac{d_i}{(x_i)^\alpha}\right)$, either $x_i > d_i^{\frac{1}{\alpha}}$, or $x_i = 0$: indeed, if we give application T_i a fraction of cache smaller than $d_i^{\frac{1}{\alpha}}$, the minimum is equal to 1, and this fraction is wasted.

Hence, we have for all i:

$$x_i = 0 \text{ or } d_i^{\frac{1}{\alpha}} < x_i \le \frac{a_i}{C_s}. \tag{3}$$

Of course, if $d_i^{\frac{1}{\alpha}} \ge \frac{a_i}{C_s}$ for some application T_i, then $x_i = 0$.

Scheduling problem. Given n applications $T_1, \dots, T_n$, we aim at partitioning the shared cache and assign processors so that the concurrent execution of these applications takes minimal time. In other words, we aim at minimizing the execution time of the longest application, when all applications start their execution at the same time. Formally:

Definition 1 (CoSchedCache). *Given n applications $T_1, \dots, T_n$ and a platform with p identical processors sharing a cache of size C_s, find a schedule $\{(p_1, x_1), \dots, (p_n, x_n)\}$ with $\sum_{i=1}^n p_i \le p$, and $\sum_{i=1}^n x_i \le 1$, that minimizes*

$$\max_{1 \le i \le n} \mathcal{E}xe_i(p_i, x_i).$$

4 Complexity Results for Perfectly Parallel Applications

In this section, we consider CoSchedCache with perfectly parallel applications. A perfectly parallel application T_i is an application with $s_i = 0$, so that $\mathcal{E}\mathrm{xe}_i(p_i, x_i) = \frac{\mathcal{E}\mathrm{xe}_i(1, x_i)}{p_i}$. Let $\mathcal{E}\mathrm{xe}_i^{\mathrm{seq}}(x_i) = \mathcal{E}\mathrm{xe}_i(1, x_i)$ be the sequential execution time of application T_i with a fraction of cache x_i. The main results are the NP-completeness of CoSchedCache, and several dominance results on the optimal solution. While these results only hold for perfectly parallel applications, they will guide the design of heuristics for general applications in Section 5.

4.1 Intractability

We formally state the decision problem associated to CoSchedCache:

Definition 2 (CoSchedCache-Dec)**.** *Given n perfectly parallel applications $T_1, \dots, T_n$ and a platform with p identical processors sharing a cache of size C_s, and given a bound K on the makespan, does there exist a schedule $\{(p_1, x_1), \dots, (p_n, x_n)\}$, where p_i and x_i are nonnegative rational numbers with $\sum_{i=1}^{n} p_i \le p$ and $\sum_{i=1}^{n} x_i \le 1$, such that $\max_{1 \le i \le n} \mathcal{E}xe_i(p_i, x_i) \le K$?*

Theorem 1. CoSchedCache-Dec *is NP-complete.*

Proof. For perfectly parallel applications, we can transform CoSchedCache into an equivalent problem that does not depend on the number of processors but that relies simply on the cache partitioning strategy (Lemma 3 below). This result will guide processor assignment for general applications in Section 5. We start with a few lemmas. The first lemma shows that all applications complete at the same time in an optimal execution:

Lemma 1. *To minimize the makespan, all applications must finish at the same time.*

Proof. Consider the n perfectly parallel applications $T_1, \dots, T_n$ and a solution $\mathcal{S} = \{(p_i, x_i)\}_{1 \le i \le n}$ to CoSchedCache. Let $D_\mathcal{S} = \max_i \mathcal{E}\text{xe}_i(p_i, x_i)$ be the makespan of this solution. We denote by $I_\mathcal{S}$ the set of applications whose execution time is exactly $D_\mathcal{S}$.

We show the result by contradiction. We consider an optimal solution $\mathcal{S}$ whose subset $I_\mathcal{S}$ has minimal size (i.e., for any other optimal solution $\mathcal{S}_o$, $|I_\mathcal{S}| \le |I_{\mathcal{S}_o}|$). Then we show that if $|I_\mathcal{S}| \ne n$, we can construct a solution $\mathcal{S}'$ with either (i) a smaller makespan if $|I_\mathcal{S}| = 1$ (contradicting the optimality hypothesis), or (ii) one less application whose execution time is exactly $D_\mathcal{S}$ (contradicting the minimality hypothesis). Assume $|I_\mathcal{S}| \ne n$, let $T_{i_0} \notin I_\mathcal{S}$ and $T_{i_1} \in I_\mathcal{S}$. We have $\mathcal{E}\text{xe}_{i_0}(p_{i_0}, x_{i_0}) < \mathcal{E}\text{xe}_{i_1}(p_{i_1}, x_{i_1}) = D_\mathcal{S}$. We define

$$\begin{aligned}\varepsilon &= p_{i_0} p_{i_1} \frac{\mathcal{E}\text{xe}_{i_1}(p_{i_1}, x_{i_1}) - \mathcal{E}\text{xe}_{i_0}(p_{i_0}, x_{i_0})}{\mathcal{E}\text{xe}^{\text{seq}}_{i_0}(x_{i_0}) + \mathcal{E}\text{xe}^{\text{seq}}_{i_1}(x_{i_1})} \\ &= \frac{p_{i_0} \mathcal{E}\text{xe}^{\text{seq}}_{i_1}(x_{i_1}) - p_{i_1} \mathcal{E}\text{xe}^{\text{seq}}_{i_0}(x_{i_0})}{\mathcal{E}\text{xe}^{\text{seq}}_{i_0}(x_{i_0}) + \mathcal{E}\text{xe}^{\text{seq}}_{i_1}(x_{i_1})} > 0.\end{aligned}$$

Then clearly, $\mathcal{S}' = \{(p'_i, x_i)\}_i$ where p'_i is (i) p_i if $i \notin \{i_0, i_1\}$, (ii) $p_{i_0} - \varepsilon$ if $i = i_0$, (iii) $p_{i_1} + \varepsilon$ if $i = i_1$, is a valid solution: we have the property $\sum_i p'_i = \sum_i p_i \le p$, and $\sum_i x'_i = \sum_i x_i \le 1$.

Furthermore, one can verify that $\mathcal{E}\text{xe}_{i_1}(p'_{i_1}, x_{i_1}) = \mathcal{E}\text{xe}_{i_0}(p'_{i_0}, x_{i_0}) = \frac{\mathcal{E}\text{xe}^{\text{seq}}_{i_0}(x_{i_0}) + \mathcal{E}\text{xe}^{\text{seq}}_{i_1}(x_{i_1})}{p_{i_0} + p_{i_1}}$. Because $p'_{i_1} > p_{i_1}$, $\mathcal{E}\text{xe}_{i_1}(p'_{i_1}, x_{i_1}) < \mathcal{E}\text{xe}_{i_1}(p_{i_1}, x_{i_1}) = D_\mathcal{S}$. Hence:

- If $|I_\mathcal{S}| = 1$, then for all i, $\mathcal{E}\text{xe}_i(p'_i, x_i) < D_\mathcal{S}$, hence showing that $\mathcal{S}$ is not optimal.
- Else, $I_{\mathcal{S}'} = I_{\mathcal{S}'} \setminus \{i_1\}$, and $D_{\mathcal{S}'} = D_\mathcal{S}$, hence showing that $\mathcal{S}$ is not minimal.

This shows that necessarily, $|I_\mathcal{S}| = n$. □

The second lemma shows the optimal processor assignment:

Lemma 2. *Given n applications $T_1, \dots, T_n$ and a partitioning of the cache $\{x_1, \dots, x_n\}$, then the optimal number of processors for application T_i ($i \in \{1, \dots, n\}$) is:*

$$p_i = p \frac{\mathcal{E}xe^{seq}_i(x_i)}{\sum_{j=1}^{n} \mathcal{E}xe^{seq}_j(x_j)}.$$

Proof. According to Lemma 1, all applications finish at the same time. Given $i_0 \in \{1, \dots, n\}$, we have $\frac{\mathcal{E}\text{xe}^{\text{seq}}_{i_0}(x_{i_0})}{p_{i_0}} = \frac{\mathcal{E}\text{xe}^{\text{seq}}_{i}(x_{i})}{p_{i}}$ for all $1 \leq i \leq n$. In addition, we have $\sum_{i=1}^{n} p_i = p$: the fact that this bound is tight in an optimal solution is due to the fact that we have perfectly parallel applications. We express p in terms of the others variables, and we do the summation: $p = \sum_{i=1}^{n} p_i = \frac{p_{i_0}}{\mathcal{E}\text{xe}^{\text{seq}}_{i_0}(x_{i_0})} \sum_{i=1}^{n} \mathcal{E}\text{xe}^{\text{seq}}_{i}(x_i)$. This directly leads to the result. □

Lemmas 1 and 2 lead to the following reformulation of CoSchedCache:

Lemma 3. CoSchedCache *can be rewritten as finding the optimal cache partitioning strategy $\mathcal{X} = \{x_1, \dots, x_n\}$ that minimizes the completion time of an optimal solution:*

$$\frac{1}{p} \sum_{i=1}^{n} \mathcal{E}xe_i(1, x_i). \tag{4}$$

Proof. Lemma 2 gives us that in an optimal solution the processor distribution is uniquely determined by the cache partitioning strategy. Furthermore, given a cache partitioning strategy, we know that all applications finish at the same time (Lemma 1) and that the completion time is equal to

$$\frac{\mathcal{E}\text{xe}^{\text{seq}}_{1}(x_1)}{p_1} = \frac{\sum_{i=1}^{n} \mathcal{E}\text{xe}^{\text{seq}}_{i}(x_i)}{p}.$$

□

We are now ready for the proof of Theorem 1. CoSchedCache-Dec is obviously in NP: given the x_i's, it is easy to verify all constraints in linear time. We prove the completeness by a reduction from Knapsack, which is NP-complete [10]. Consider an arbitrary instance $\mathcal{I}_1$ of Knapsack: given n objects, each with positive integer size u_i and positive integer value v_i for $1 \leq i \leq n$, and two positive integer bounds U and V, does there exist a subset $I \subset \{1, \dots, n\}$ such that $\sum_{i \in I} u_i \leq U$ and $\sum_{i \in I} v_i \geq V$? Given $\mathcal{I}_1$, we construct the following instance $\mathcal{I}_2$ of CoSchedCache-Dec:

- We define two constants $\varepsilon = \frac{1}{N(N+1)}$ and $\eta = 1 - \frac{1}{N}$, where $N = \max(n, 2U + 1)$.
- We let $d_i = \left(\frac{u_i \eta}{U}\right)^{\alpha}$, $e_i = \left(d_i^{\frac{1}{\alpha}} + \varepsilon\right)^{\alpha}$, $a_i = e_i^{\frac{1}{\alpha}} C_s$, and $w_i f_i l_l = \frac{v_i}{1 - \frac{d_i}{e_i}}$ for $1 \leq i \leq n$. Note that we only need the value of the product $w_i f_i$, and we can set one of them arbitrarily.
- The bound K is defined as:

$$pK = \sum_{i=1}^{n} w_i (1 + f_i l_s) + \sum_{i=1}^{n} w_i f_i l_l - V.$$

To simplify notations, let $z_i = w_i f_i l_l$. Letting $A = \sum_{i=1}^{n} w_i (1 + f_i l_s)$ and $Z = \sum_{i=1}^{n} z_i$, we get $pK = A + Z - V$. Also, we have $\sum_{i=1}^{n} w_i \left(1 + f_i \left[l_s + l_l \cdot \min\left(1, \frac{d_i}{x_i^{\alpha}}\right)\right]\right) = A + B$, where $B = \sum_{i=1}^{n} z_i \min(1, \frac{d_i}{x_i^{\alpha}})$. Recall from Lemma 3 that $\mathcal{I}_2$ has a solution if and only if $\frac{1}{p}(A+B) \leq K$.

Let $I_C \subseteq \{1, \dots, n\}$ denote the subset of applications that are given some cache ($x_i \neq 0$ if and only if $i \in I_C$). We also call I_C the nonzero subset of $\mathcal{I}_2$. We have

$$d_i^{\frac{1}{\alpha}} \leq x_i \leq \frac{a_i}{C_s} = e_i^{\frac{1}{\alpha}},$$

so that we can rewrite $B = Z - \sum_{i \in I_C} z_i \left(1 - \frac{d_i}{x_i^{\alpha}}\right)$. Given the value of the bound K, we have $A + B \leq pK$ if and only if

$$\sum_{i \in I_C} z_i (1 - \frac{d_i}{x_i^{\alpha}}) \geq V.$$

We show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does. Suppose first that $\mathcal{I}_1$ has a solution subset $I \subset \{1, \dots, n\}$. Then we let $x_i = e_i^{\frac{1}{\alpha}}$ if $i \in I$ and $x_i = 0$ otherwise. This is a valid solution to $\mathcal{I}_2$ with nonzero subset $I_C = I$. Indeed:

- If $i \in I$, then $d_i^{\frac{1}{\alpha}} \le x_i = e_i^{\frac{1}{\alpha}} = \frac{a_i}{C_s}$.
- We have
$$\sum_{i \in I} x_i = \sum_{i \in I} (d_i^{\frac{1}{\alpha}} + \varepsilon) = \sum_{i \in I} \frac{u_i \eta}{U} + |I|\varepsilon.$$
But $\sum_{i \in I} \frac{u_i \eta}{U} \le \eta$ (since we have a solution for $\mathcal{I}_1$), and $|I|\varepsilon \le n\varepsilon \le \frac{1}{N+1}$, hence $\sum_{i \in I} x_i \le \eta + \frac{1}{N+1} \le 1$.
- Finally, $\sum_{i \in I} z_i(1 - \frac{d_i}{x_i^\alpha}) = \sum_{i \in I} z_i(1 - \frac{d_i}{e_i}) = \sum_{i \in I} v_i \ge V$ (since we have a solution for $\mathcal{I}_1$), hence $A + B \le pK$.

Suppose now that $\mathcal{I}_2$ has a solution, and let I_C be its nonzero subset. We claim that $I = I_C$ is a solution to $\mathcal{I}_1$. Indeed, for $i \in I_C$ we have $d_i \le x_i^\alpha \le e_i$ and $\sum_{i \in I_C} z_i(1 - \frac{d_i}{x_i^\alpha}) \ge V$. First, we have $\sum_{i \in I_C} z_i(1 - \frac{d_i}{x_i^\alpha}) \ge \sum_{i \in I_C} z_i(1 - \frac{d_i}{e_i}) = \sum_{i \in I_C} v_i$, hence $\sum_{i \in I_C} v_i \ge V$. Then $\sum_{i \in I_C} d_i^{\frac{1}{\alpha}} \le \sum_{i \in I_C} x_i \le 1$, and $\sum_{i \in I_C} d_i^{\frac{1}{\alpha}} = \sum_{i \in I_C} \frac{u_i \eta}{U}$, hence $\sum_{i \in I_C} u_i \le \frac{U}{\eta}$. But $\frac{U}{\eta} \le U + \frac{1}{2}$ by the choice of η, thus $\sum_{i \in I_C} u_i \le U + \frac{1}{2}$. Because the sizes are integers, $\sum_{i \in I_C} u_i \le U$. Altogether, I_C is indeed a solution to $\mathcal{I}_1$. This concludes the proof. □

4.2 Dominance results

In this section, we provide dominance results that will guide the design of heuristics. In addition to restricting to perfectly parallel applications ($s_i = 0$), we assume that application memory footprints are larger than the cache size ($a_i = +\infty$). The core of the previous intractability result relies on the hardness to determine the set of applications that receive a cache fraction (denoted by I_C) and those that do not (denoted by $\overline{I_C}$). In this section, we show (i) how to determine the optimal solution when these sets I_C and $\overline{I_C}$ are known, and (ii) whether one can disqualify some partitions as being sub-optimal.

In particular, we define a set of partitions of applications that we call dominant (Definition 4). We show that (i) if a partition of applications $I_C, \overline{I_C}$ is dominant, then we can compute the minimum execution time for this partition, and (ii) if a partition is not dominant, then we can find a better dominant partition. We start by rewriting the problem when the partitioning $I_C, \overline{I_C}$ of applications is known:

Definition 3 (CoSchedCache-Part$(I_C, \overline{I_C})$)**.** *Given a set of applications $T_1, \dots, T_n$ and a partition $I_C, \overline{I_C}$,* CoSchedCache-Part$(I_C, \overline{I_C})$ *is the problem to find a set $\mathcal{X} = \{x_1, \dots, x_n\}$ that minimizes the execution time:*

$$\frac{1}{p}\left(\sum_{i \in \overline{I_C}} w_i(1 + f_i(l_s + l_l)) + \sum_{i \in I_C} w_i(1 + f_i l_s + f_i l_l \frac{d_i}{x_i^\alpha})\right)$$

under the constraints $x_i = 0$ if $i \in \overline{I_C}$, $x_i > d_i^{1/\alpha}$ if $i \in I_C$, and $\sum_{1 \le i \le n} x_i \le 1$.

We now relax some bounds in CoSchedCache-Part$(I_C, \overline{I_C})$ and define CoSchedCache-Ext$(I_C, \overline{I_C})$, which is the same problem except that the constraints on the x_i's when $i \in I_C$ is relaxed: we have instead $x_i \ge 0$ if $i \in I_C$.

A solution of CoSchedCache-Part$\left(I_C, \overline{I_C}\right)$ is a solution of CoSchedCache-Ext$\left(I_C, \overline{I_C}\right)$, because we simply removed the constraints $x_i > d_i^{1/\alpha}$ in the latter problem. Hence the execution time of the optimal solution of CoSchedCache-Ext$\left(I_C, \overline{I_C}\right)$ is lower than that of CoSchedCache-Part$\left(I_C, \overline{I_C}\right)$. Furthermore, given a solution of CoSchedCache-Ext$\left(I_C, \overline{I_C}\right)$, one can easily see that its execution time in CoSchedCache will be lower (the objective function is lower since it involves a minimum for all applications in I_C).

Lemma 4. *Given a set of applications $T_1, \ldots, T_n$ and a partition $I_C, \overline{I_C}$, the optimal solution to* CoSchedCache-Ext$\left(I_C, \overline{I_C}\right)$ *is*

$$\begin{aligned} x_i &= \frac{\left(w_i f_i d_i\right)^{1/(\alpha+1)}}{\sum_{j \in I_C} \left(w_j f_j d_j\right)^{1/(\alpha+1)}} && \text{if } i \in I_C, \\ x_i &= 0 && \text{otherwise.} \end{aligned}$$

Proof. We want to compute $\mathcal{X} = \{x_1, \ldots, x_n\}$ that minimizes the execution time. Discarding constant factors, this reduces to minimizing

$$K(\mathcal{X}) = \sum_{i \in I_C} \frac{w_i f_i d_i}{x_i^\alpha}$$

under the constraints: $x_i = 0$ if $i \in \overline{I_C}$, $x_i \geq 0$ otherwise, and $\sum_i x_i \leq 1$. Clearly, one can see that this last inequality is an equality when $I_C \neq \emptyset$ (otherwise K is not minimum).

To minimize the function, we compute the partial derivatives of K:

$$\forall i \in I_C, \; \frac{\partial K(\mathcal{X})}{\partial x_i} = -\alpha \frac{w_i f_i d_i}{x_i^{\alpha+1}}.$$

By setting them all to 0, we obtain the following equality for $1 \leq i \leq n$:

$$-\alpha \frac{w_i f_i d_i}{x_i^{\alpha+1}} = -\alpha \frac{w_n f_n d_n}{x_n^{\alpha+1}}.$$

Hence,

$$\begin{aligned} \forall i \in I_C, \; x_i &= x_n \frac{\left(w_i f_i d_i\right)^{\frac{1}{\alpha+1}}}{\left(w_n f_n d_n\right)^{\frac{1}{\alpha+1}}}; \\ \sum_{i=1}^{n} x_i &= \frac{x_n}{\left(w_n f_n d_n\right)^{\frac{1}{\alpha+1}}} \sum_{i \in I_C} \left(w_i f_i d_i\right)^{\frac{1}{\alpha+1}} \\ &= 1. \end{aligned}$$

Hence, the desired result. □

Definition 4 (Dominant partition)**.** *Given a set of applications $T_1, \ldots, T_n$, we say that a partition of these applications $I_C, \overline{I_C}$ is* dominant*, if for all $i \in I_C$,*

$$\frac{\left(w_i f_i d_i\right)^{1/(\alpha+1)}}{\sum_{j \in I_C} \left(w_j f_j d_j\right)^{1/(\alpha+1)}} > d_i^{1/\alpha}.$$

We can now state the following result:

Theorem 2. *If a partition $I_C, \overline{I_C}$ is not dominant, then we can compute in polynomial time a better solution.*

Proof. Let $I_C, \overline{I_C}$ be a non-dominant partition.

Let $i_0 \in I_C$ such that $\frac{\left(w_{i_0} f_{i_0} d_{i_0}\right)^{1/(\alpha+1)}}{\sum_{j\in I_C} (w_j f_j d_j)^{1/(\alpha+1)}} \le d_{i_0}^{1/\alpha}$.

First we can show that there is $i_1 \in I_C \backslash \{i_0\}$. Indeed, otherwise we would have $\frac{\left(w_{i_0} f_{i_0} d_{i_0}\right)^{1/(\alpha+1)}}{\sum_{j\in I_C} (w_j f_j d_j)^{1/(\alpha+1)}} = 1 \le d_{i_0}^{1/\alpha}$, and $I_C, \overline{I_C}$ is not a valid partition: then CoSchedCache-Part$\left(I_C, \overline{I_C}\right)$ does not admit any solution.

Let $\mathcal{T}_e$ (resp. $\mathcal{T}_p$) be the optimal execution time of CoSchedCache-Ext$\left(I_C, \overline{I_C}\right)$ (resp. CoSchedCache-Part$\left(I_C, \overline{I_C}\right)$). We know that $\mathcal{T}_e \le \mathcal{T}_p$. Let us further denote by $\mathcal{X} = \{x_1, \ldots, x_n\}$ the optimal solution to CoSchedCache-Ext$\left(I_C, \overline{I_C}\right)$. Let $\bar{\mathcal{X}} = \{\bar{x}_1, \ldots, \bar{x}_n\}$ be such that (i) $\bar{x}_{i_0} = 0$, (ii) $\bar{x}_{i_1} = x_{i_0} + x_{i_1}$, and (iii) $\bar{x}_i = x_i$ for all other i's.

Then clearly $\bar{\mathcal{X}}$ is a solution, and we have:

$$
\begin{aligned}
&\mathcal{E}\mathrm{xe}_{i_0}^{\mathrm{seq}}(\bar{x}_{i_0}) \le w_{i_0}\left(1 + f_{i_0} l_s + f_{i_0} l_l \frac{d_{i_0}}{x_{i_0}^{\alpha}}\right); \\
&\mathcal{E}\mathrm{xe}_{i_1}^{\mathrm{seq}}(\bar{x}_{i_1}) < w_{i_1}\left(1 + f_{i_1} l_s + f_{i_1} l_l \frac{d_{i_0}}{x_{i_1}^{\alpha}}\right); && (5)\\
&\mathcal{E}\mathrm{xe}_{i}^{\mathrm{seq}}(\bar{x}_{i}) \le w_{i}\left(1 + f_{i} l_s + f_{i} l_l \frac{d_{i}}{x_{i}^{\alpha}}\right) && \text{if } i \in I_C; \\
&\mathcal{E}\mathrm{xe}_{i}^{\mathrm{seq}}(\bar{x}_{i}) = w_i\left(1 + f_i(l_s + l_l)\right) && \text{if } i \in \overline{I_C}.
\end{aligned}
$$

Indeed, these results are direct consequences of the definition of $\mathcal{E}\mathrm{xe}^{\mathrm{seq}}$, except Equation (5), which we establish as follows:

- If $x_{i_1} \ge d_{i_1}^{1/\alpha}$, then $\bar{x}_{i_1} > d_{i_1}^{1/\alpha}$

$$
\begin{aligned}
\mathcal{E}\mathrm{xe}_{i_1}^{\mathrm{seq}}(\bar{x}_{i_1}) &= w_{i_1}\left(1 + f_{i_1} l_s + f_{i_1} l_l \frac{d_{i_0}}{\bar{x}_{i_1}^{\alpha}}\right) \\
&< w_{i_1}\left(1 + f_{i_1} l_s + f_{i_1} l_l \frac{d_{i_0}}{x_{i_1}^{\alpha}}\right).
\end{aligned}
$$

- If $x_{i_1} < d_{i_1}^{1/\alpha}$, then for all $x \in [0,1]$, $\mathcal{E}\mathrm{xe}_{i_1}^{\mathrm{seq}}(x) < w_{i_1}\left(1 + f_{i_1} l_s + f_{i_1} l_l \frac{d_{i_0}}{x_{i_1}^{\alpha}}\right)$.

Hence:

$$
\begin{aligned}
\frac{1}{p}\sum_{i=1}^{n} \mathcal{E}\mathrm{xe}_i^{\mathrm{seq}}(\bar{x}_i) &< \frac{1}{p}\Big(\sum_{i\in\overline{I_C}} w_i(1 + f_i(l_s + l_l)) \\
&\quad + \sum_{i\in I_C} w_i(1 + f_i l_s + f_i l_l \frac{d_i}{x_i^{\alpha}})\Big) = \mathcal{T}_e \le \mathcal{T}_p,
\end{aligned}
$$

which shows that $\bar{\mathcal{X}}$ is a better solution computed in polynomial time from $\mathcal{X}$. Furthermore, by construction of $\bar{\mathcal{X}}$, we have strictly decreased the size of the new set I_C. □

Finally, we can show a second dominance result:

Theorem 3. *If a partition $I_C, \overline{I_C}$ is dominant, then the optimal solution to* CoSchedCache-Part$\left(I_C, \overline{I_C}\right)$ *is:*

$$
\begin{aligned}
x_i &= \frac{(w_i f_i d_i)^{1/(\alpha+1)}}{\sum_{j\in I_C} (w_j f_j d_j)^{1/(\alpha+1)}} && \textit{if } i \in I_C; \\
x_i &= 0 && \textit{otherwise.}
\end{aligned}
$$

Proof. This is a corollary of Lemma 4, this solution is the optimal solution to CoSchedCache-Ext$(I_C, \overline{I_C})$ and it is a valid solution to CoSchedCache-Part$(I_C, \overline{I_C})$, hence it is the optimal solution to CoSchedCache-Part$(I_C, \overline{I_C})$. □

5 Heuristics

In this section, we aim at designing efficient heuristics for general applications that obey Amdahl's law, and whose memory footprints are larger than the cache size ($a_i = +\infty$). However, the CoSchedCache problem seems to be very difficult for such applications. In particular, Lemma 2 does not hold, and we have no guideline to decide how many processors to assign. We simplify the design of the heuristics by temporarily allocating processors as if the applications were perfectly parallel, and then concentrating on strategies that partition the cache efficiently among some applications (and give no cache fraction to remaining ones). In accordance with Theorem 2, our goal is to compute dominant partitions. Recall that I_C represents the subset of applications that receive a fraction fo the cache. Once a dominant partition is given, we obtain the schedule $\mathcal{S} = \{(x_i, p_i)\}_i$ as follows: first we determine the x_i's with Theorem 3, and then we recompute the p_i's so that all applications complete simultaneously at time K. However, there is no longer a nice analytical characterization of the makespan K, hence we use a binary search to compute K as follows: for each application T_i, the execution time writes $(s_i + \frac{1-s_i}{p_i})c_i = K$, where s_i is the sequential fraction, and $c_i = w_i(1 + f_i(l_s + l_l \frac{d_i}{x_i^\alpha}))$ if $T_i \in I_C$, or $c_i = w_i(1 + f_i(l_s + l_l))$ otherwise. From $\sum_{i=1}^n p_i = p$, we derive the equation

$$\sum_{i=1}^{n} \frac{1 - s_i}{\frac{K}{c_i} - s_i} = p$$

and we compute K through a binary search. A lower (resp. upper) bound for K is to assign p (resp. 1) processor(s) to each application.

To compute dominant partitions, we use two greedy strategies:

- Dominant: we start with $I_C = \mathcal{I}$ and greedily remove some applications from I_C until we have a dominant partition (see Algorithm 1);
- DominantRev: initially I_C is empty, and we greedily add applications while I_C remains dominant (see Algorithm 2).

In both strategies, the greedy criterion to select the next application is the *choice* function taken from the following three alternatives:

- Random: *choice*$(\mathcal{I})$ picks up randomly one application among all applications;

Algorithm 1: Dominant partition, starting with all applications

procedure Dominant ($\mathcal{I}$, *choice*) **begin**
    $I_C \leftarrow \mathcal{I}$;
    while $\exists i \in I_C$ *s.t.* $\frac{(w_i f_i d_i)^{1/(\alpha+1)}}{d_i^{1/\alpha}} \geq \sum_{j \in I_C} (w_j f_j d_j)^{1/(\alpha+1)}$ **do**
        $k \leftarrow$ *choice*(I_C);
        $I_C \leftarrow I_C \setminus \{k\}$;
        if $I_C = \emptyset$ **then break**;
    end
    $\overline{I_C} \leftarrow \mathcal{I} \setminus I_C$;
    return $(I_C, \overline{I_C})$;
end

Algorithm 2: Dominant partition, starting from empty set

procedure DominantRev ($\mathcal{I}$, *choice*) **begin**
  $\overline{I_C} \leftarrow \mathcal{I}$; $I_C \leftarrow \emptyset$;
  $k \leftarrow choice(\overline{I_C})$;
  $I'_C \leftarrow \{k\}$;
  while $\forall i \in I'_C \ \frac{(w_i f_i d_i)^{1/(\alpha+1)}}{d_i^{1/\alpha}} < \sum_{j \in I'_C} (w_j f_j d_j)^{1/(\alpha+1)}$ **do**
    $I_C \leftarrow I'_C$;
    $\overline{I_C} \leftarrow \overline{I_C} \setminus \{k\}$;
    if $\overline{I_C} = \emptyset$ **then break**;
    $k \leftarrow choice(\overline{I_C})$;
    $I'_C \leftarrow I'_C \cup \{k\}$;
  end
  return $(I_C, \overline{I_C})$;
end

• MinRatio considers the ratio that appears in Definition 4 (dominant partitions), and chooses an application with a small ratio:

$$choice(\mathcal{I}) = \arg\min_{i \in \mathcal{I}} \left(\frac{(w_i f_i d_i)^{1/(\alpha+1)}}{d_i^{1/\alpha}} \right);$$

• MaxRatio proceeds the other way round, by choosing an application with a large ratio:

$$choice(\mathcal{I}) = \arg\max_{i \in \mathcal{I}} \left(\frac{(w_i f_i d_i)^{1/(\alpha+1)}}{d_i^{1/\alpha}} \right).$$

The intuition behind these heuristics is the following: applications that render a solution non dominant are such that (see Definition 4):

$$\frac{(w_i f_i d_i)^{1/(\alpha+1)}}{d_i^{1/\alpha}} \leq \sum_{j \in I_C} (w_j f_j d_j)^{1/(\alpha+1)}.$$

Hence, we expect to reach dominance faster by removing from a non-dominant solution applications with low $\frac{(w_i f_i d_i)^{1/(\alpha+1)}}{d_i^{1/\alpha}}$ (left term of the equation). Intuitively, Dominant should work well with the MinRatio criterion. For symmetric reasons, we expect DominantRev to work well with the MaxRatio criterion. These intuitions will be experimentally confirmed in Section 6. Altogether, by combining two strategies and three different *choice* functions, we obtain six heuristics to build dominant partitions.

6 Simulations

To assess the efficiency of the heuristics defined in Section 5, we have performed extensive simulations. The simulation settings are discussed in Section 6.1, and results are presented in Sections 6.2 and 6.3. The code is publicly available at `http://perso.ens-lyon.fr/loic.pottier/archives/simu-cache.zip`.

6.1 Simulation settings

We use data from applicative benchmarks to run the experiments. Table 1 provides a brief description of the NAS Parallel Benchmark (NPB) suite [3], and Table 2 shows the parameters

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

Table 1: Description of the NPB benchmarks.

App	w_i	f_i	$m^i_{40\text{MB}\mathcal{S}_s}$
CG	5.70E+10	5.35E-01	6.59E-04
BT	2.10E+11	8.29E-01	7.31E-03
LU	1.52E+11	7.50E-01	1.51E-03
SP	1.38E+11	7.62E-01	1.51E-02
MG	1.23E+10	5.40E-01	2.62E-02
FT	1.65E+10	5.82E-01	1.78E-02

Table 2: Experimental values from NPB benchmarks.

for these six HPC applications. We obtain the values shown in Table 2 by instrumenting and simulating the benchmarks (*CLASS=A*) on 16 cores using PEBIL [18]. For the simulations, we use a cache configuration representing an Intel Xeon CPU E5-2690, with a 20MB last level cache per processor of 8 cores. Since the cache miss ratio is defined for a 40MB cache, we have $d_i = m^i_{40\text{MB}\mathcal{S}_s}\left(\frac{40\times 10^6}{C_s}\right)^\alpha$.

We build synthetic applications from Table 2 by varying the work w_i randomly between 1E+8 and 1E+12. Other data sets building upon these applications have been used (see the Appendix A), and the results are very similar. The sequential fraction of work s_i is taken randomly between 1% and 15%.

For the execution platform, we consider one manycore *Sunway TaihuLight* [7] with 256 processors and a shared memory of 32GB. We chose this platform because of its high core count. Strictly speaking, this platform does not have a last level cache (LLC), but the shared memory can be seen as the LLC, using the disk as the large memory. We have $C_s = 32000 \times 10^6$. The large storage latency l_l is set to 1. The small storage latency l_s is set to 0.17. According to the literature [17, 21, 23], the last level cache (LLC) latency is on average four to ten times better

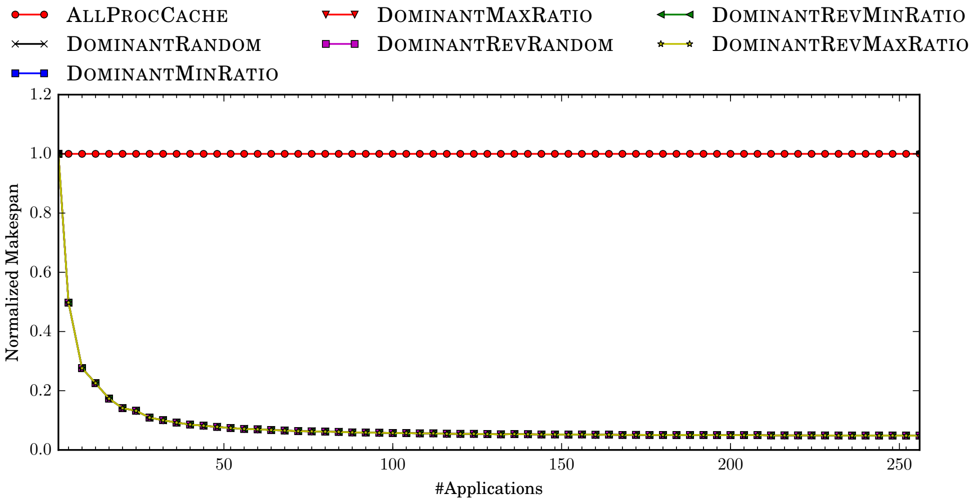

Figure 1: Comparison of the six dominant partition heuristics.

than the DDR latency, and we enforce a ratio of 5.88 in the simulations. We have used different ratios in Appendix A, and they lead to similar results. Finally, the Power Law parameter is fixed to $\alpha = 0.5$. We execute each heuristic 50 times and we compute the average *makespan*, i.e., the longest execution time among all co-scheduled applications.

In Section 6.2, we provide a comparison of the six heuristics of Section 5, before assessing the gain due to co-scheduling in Section 6.3.

6.2 Comparison of the six heuristics

Figure 1 shows the normalized makespan obtained by the six variants of heuristics building dominant partitions. Results are normalized with the makespan of AllProcCache, which is the execution without any co-scheduling: in the AllProcCache heuristic, applications are executed sequentially, each using all processors and all the cache. We vary the number of applications from 1 to 256 on 256 processors. The six heuristics obtain similarly good results, with a gain of 85% over AllProcCache as soon as there are at least 50 applications.

Since all six variants show the same performance on the previous data sets, we investigate the impact of the cache miss rate by varying it between 0 and 1 with a LLC of $C_s = 1GB$ in Figure 2. Results are now normalized with DominantMinRatio, which enables to zoom out the differences. Actually, DominantMinRatio and DominantRevMaxRatio are always the best (their plots overlap), while DominantMaxRatio and DominantRevMinRatio are almost always the worst (again, their plots overlap), hence confirming the intuition in Section 5. The random variants lay in between. We use a 1GB LLC to show the impact of cache miss rate on heuristic behaviors. We observe that there are some differences between heuristics only when the cache miss ratio becomes greater than 0.1. According to current data, $m_{40\text{MB}\mathcal{S}_s}$ range from 1E-02 to 1E-04. In addition, these differences are visible only with a small shared memory (1GB in the example) while our execution platform has a 32GB shared memory. Overall, for the system used in these simulations, all heuristics perform similarly, even though DominantMinRatio and DominantRevMaxRatio seem to perform best in all other settings that we tried (see Appendix A). Therefore, for clarity, we plot only one heuristic based on dominant partitions in the remaining simulations, namely DominantMinRatio.

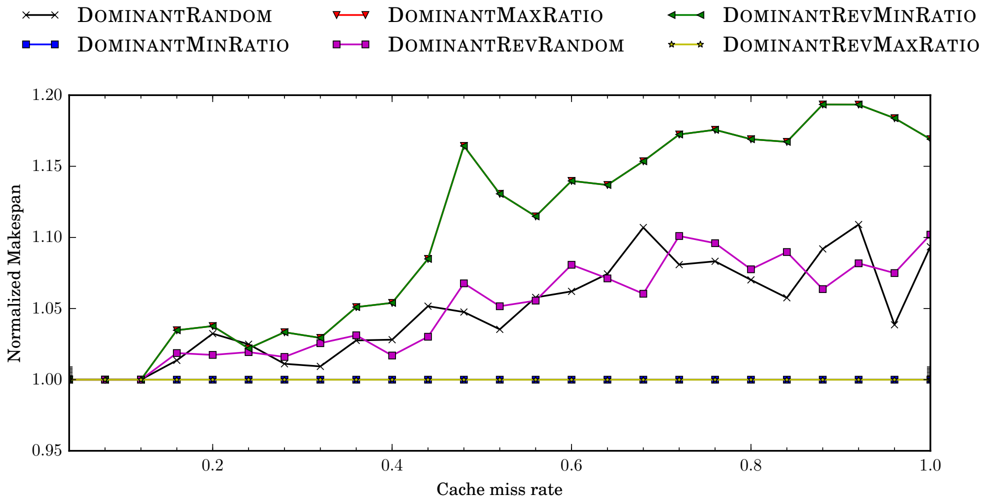

Figure 2: Impact of cache miss rate using a 1GB LLC.

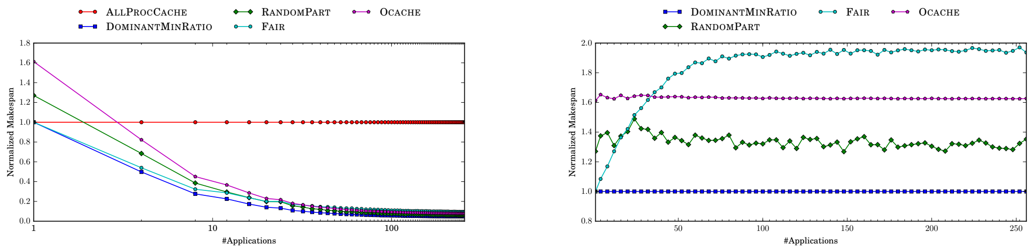

Figure 3: Impact of the number of applications.

6.3 Gain with co-scheduling

In this section, we assess the gain due to co-scheduling by comparing DominantMinRatio with AllProcCache and with three other heuristics:

- Fair gives $p_i = \frac{p}{n}$ processors, and a fraction of cache $x_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$ to each application;
- 0cache gives no cache to any application, i.e., $x_i = 0$ for $1 \leq i \leq n$, and then it computes the p_i's so that all applications finish at the same time;
- RandomPart randomly partitions applications with and without cache. For those in cache, the x_i's are computed with the method used for dominant partitions. Then, the p_i's are computed so that all applications finish at the same time.

Impact of the number of applications. Figure 3 (normalized with AllProcCache) shows the impact of the number of applications when the number of processors is fixed to 256. We see that DominantMinRatio outperforms the other heuristics, hence showing the efficiency of our approach based on dominant partitions. Results are also normalized with DominantMinRatio, so that we can better observe the differences between co-scheduling heuristics. Fair exhibits good results only for a small number of applications, when all applications can fit into cache. Otherwise, the use of dominant partitions is much more efficient, as seen with RandomPart, or even 0cache that does not use cache but ensures that all applications finish at the same time.

Figure 4 shows the impact of the average number of processors per application. Here, results

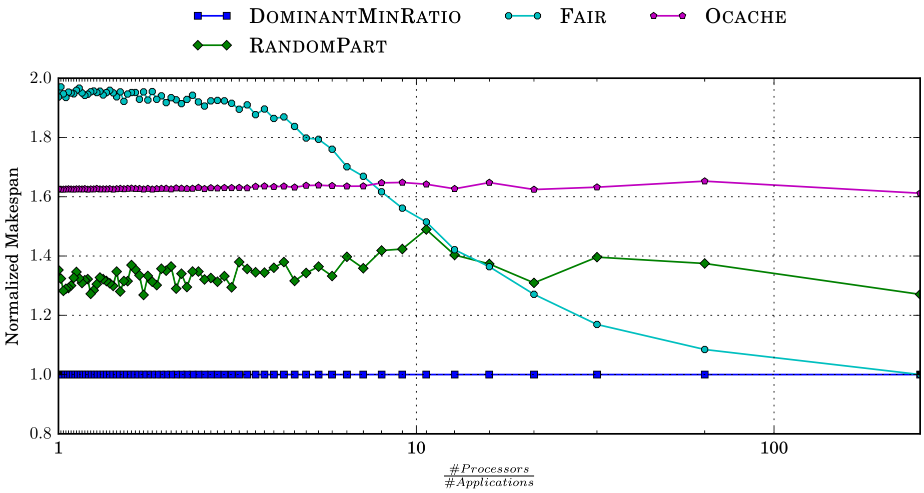

Figure 4: Impact of the average number of processors per applications.

are normalized with DominantMinRatio again, and they are similar. In particular, we see that 0cache is better than Fair when there are many processors to share between applications, even though it is not using the cache at all. These results show the accuracy of the model and the benefits of using dominant partitions. Finally, we note the importance of cache partitioning, since the difference between 0cache and DominantMinRatio relies on cache allocation.

Impact of the number of processors. Figure 5 (normalized with AllProcCache) shows the impact of the number of processors when the number of applications is fixed to 16. When the number of processors increases, the gain of co-scheduling increases. DominantMinRatio is clearly the best heuristic. RandomPart, which builds a random partition instead of a dominant one, is outperformed by DominantMinRatio, and the latter is the only heuristic that surpasses AllProcCache when the number of processors is low. So, building a dominant partition seems a good strategy to optimize the makespan.

The normalization with DominantMinRatio shows that when the number of processors increases, Fair becomes better, while RandomPart and 0cache are quite stable since they are based on the same model as DominantMinRatio. The only difference between 0cache and DominantMinRatio is the cache allocation strategy, and the gain from cleverly distributing cache fractions across applications exceeds 20%. With more applications, we obtain the same ranking of heuristics, except that Fair is always the worst heuristic: since there are less processors on average per application, a good co-scheduling policy is necessary (see Appendix A for detailed results).

Impact of the sequential fraction of work. Figure 6 (normalized with AllProcCache) shows the impact of the sequential part s_i when the number of processors is fixed to 256. The number of applications is fixed to 16. As expected, when the sequential fraction of work increases, all co-scheduling heuristics perform better than AllProcCache, and DominantMinRatio is always the best heuristic. It leads to a gain of more than 50% when $s_i = 0.01$.

The normalization with DominantMinRatio better shows the impact of the sequential part: we observe that when the sequential fraction of work increases, Fair obtains results closer to DominantMinRatio.

Summary. To summarize, all heuristics based on dominant partitions are very efficient,

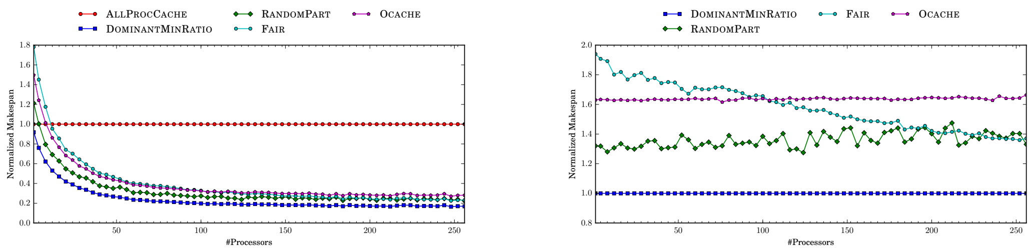

Figure 5: Impact of the number of processors.

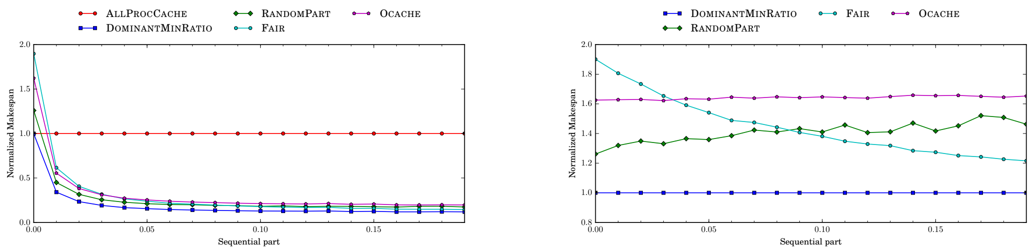

Figure 6: Impact of sequential fraction of work.

especially when compared to the classical heuristics Fair (which share the cache fairly between applications) and AllProcCache (which does no co-scheduling). The unexpected result that can be observed is that the gain brought by our heuristics comes even with very low sequential time (below 0.01)! This is unexpected since the natural intuition would be a behavior such as the one observed on Fair: a makespan up to 1.9 times longer than AllProcCache with low sequential time.

We show that the ratio processors/applications has a significant impact on performance: when many processors are available for a few applications, it is less crucial to use efficient cache-partitioning and all applications can share the cache, hence Fair obtains good results, close to DominantMinRatio. Otherwise, RandomPart is the second best heuristic, followed by 0Cache that does not use the cache, then Fair and AllProcCache.

All heuristics run within a very small time (less than ten seconds in the worst of the settings used, to be compared with a typical application execution time in hours or days), hence they can be used in practice with a very light overhead.

7 Conclusion

In this paper, we have provided a preliminary study on co-scheduling algorithms for cache-partitioned systems, building upon a theoretical study for perfectly parallel applications. The two key scheduling questions are (i) which proportion of cache and (ii) how many processors should be given to each application. We proved that the problem is NP-complete, but we have been able to characterize optimal solutions for perfectly parallel applications by introducing the concept of *dominant partitions*: for such applications, we have computed the optimal proportion of cache to give to each application in the partition. Furthermore, we have provided explicit formulas to express the number of processors to assign to each application.

Several polynomial-time heuristics have been built upon these results, and extensive simulation results demonstrated that the use of dominant partitions always leads to better results than more naive approaches, as soon as there is a small sequential fraction of work in application speedup profiles. The concept of sharing the cache only between a subset of applications seems highly relevant, since even an approach with a random selection of applications that share the cache leads to good results. Also, a clever partitioning of the cache pays off quite well, since our heuristics lead to a significant gain compared to an approach where no cache is given to applications. Overall, the heuristics appear to be very useful for general applications, even though their cache allocation strategy relies on simulating a perfectly parallel profile.

Future work will be devoted to extending the heuristics that account for the speedup profile for both processor and cache allocation. Also, we would like to gain access to, and conduct real experiments on, a cache-partitioned system with a high core count: this would allow us to further validate the accuracy of the model and to confirm the impact of our promising results.

References

[1] Advanced Scientific Computing Advisory Committee (ASCAC), "Ten technical approaches to address the challenges of Exascale computing," http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf.

[2] G. Amdahl, "The validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS Conference Proceedings*, vol. 30. AFIPS Press, 1967, pp. 483–485.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks&Mdash;Summary and Preliminary Results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 158–165. [Online]. Available: http://doi.acm.org/10.1145/125826.125925

[4] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Trans. Comput. Syst.*, vol. 28, no. 4, pp. 8:1–8:45, 2010.

[5] 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.

[6] D. Dauwe, E. Jonardi, R. Friese, S. Pasricha, A. A. Maciejewski, D. A. Bader, and H. J. Siegel, "A methodology for co-location aware application performance modeling in multicore computing," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW).* IEEE, 2015, pp. 434–443.

[7] J. Dongarra, "Report on the sunway taihulight system," *PDF). www. netlib. org. Retrieved June*, vol. 20, 2016.

[8] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei, "A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads," in *Proc. Int. conf. High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. IEEE Computer Society, 2012, pp. 83:1–83:11.

[9] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the I/O of HPC applications under congestion," in *IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*, 2015, pp. 1013–1022.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[11] 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.

[12] 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.

[13] L. He, H. Zhu, and S. A. Jarvis, "Developing graph-based co-scheduling algorithms on multicore computers," *IEEE Trans. Parallel Distributed Systems*, vol. 27, no. 6, pp. 1617–1632, 2016.

[14] Intel, "Intel 64 and IA-32 architectures software developer's manual," *Part 2*, vol. 3B: System Programming Guide, 2014.

[15] Y. Jiang, X. Shen, J. Chen, and R. Tripathi, "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *Proc. 17th Int. Conf. Parallel Architectures Compilation Techniques*, ser. PACT '08. ACM, 2008, pp. 220–229.

[16] 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.

[17] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *IEEE Int. Symp. on Performance Analysis of Systems and Software (ISPASS)*, April 2013, pp. 256–267.

[18] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "PEBIL: Efficient static binary instrumentation for Linux," in *IEEE Int. Symp. on Performance Analysis of Systems Software (ISPASS)*, March 2010, pp. 175–183.

[19] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the Ninth European Conference on Computer Systems.* ACM, 2014, p. 4.

[20] 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, p. 6, 2016.

[21] D. Molka, D. Hackenberg, R. Schone, and W. E. Nagel, "Cache coherence protocol and memory performance of the intel haswell-ep architecture," in *Int. Conf. onParallel Processing (ICPP)*, Sept 2015, pp. 739–748.

[22] 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.

[23] A. J. Pena and P. Balaji, "Toward the efficient use of multiple explicitly managed memory subsystems," in *IEEE Int. Conf. on Cluster Computing (CLUSTER)*, Sept 2014, pp. 123–131.

[24] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. 39th IEEE/ACM Int. Symp. Microarchitecture*, ser. MICRO 39. IEEE Computer Society, 2006, pp. 423–432.

[25] 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," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 371–382.

[26] C. Sewell, K. Heitmann, H. Finkel, G. Zagaris, S. T. Parete-Koon, P. K. Fasel, A. Pope, N. Frontiere, L.-t. Lo, B. Messer *et al.*, "Large-scale compute-intensive analysis via a combined in-situ and co-scheduling workflow approach," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'15*. ACM, 2015, p. 50.

[27] 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.

[28] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "Smite: Precise QOSprediction on real-system SMT processors to improve utilization in warehouse scale computers," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 406–418.

[29] H. Zhu, L. He, B. Gao, K. Li, J. Sun, H. Chen, and K. Li, "Modelling and developing co-scheduling strategies on multicore processors," in *44th Int. Conf. Parallel Processing (ICPP)*. IEEE Computer Society, 2015, pp. 220–229.

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

A Additional simulation results

We consider three sets of data for simulations:

- NPB-6: Limited to the six applications defined in Table 2;
- NPB-SYNTH: We build synthetic applications from Table 2 with only varying randomly the work w_i between 1E+8 and 1E+12 (used in the core of the paper);
- RANDOM: We build synthetic applications from Table 2 with varying all values randomly. The work w_i is taken between 1E+8 and 1E+12, f_i between 1E-01 and 9E-01, and $m^i_{40\text{MB}\mathcal{S}_s}$ between 1E-02 and 9E-04.

A.1 Impact of the number of applications

Figure 7 (normalized with AllProcCache and DominantMinRatio) shows the impact of the number of applications when the number of processors is fixed to 256. We observe similar results with RANDOM and NPB-SYNTH. Dominant partition heuristics still outperform other heuristics. As in Section 6, results are also normalized with DominantMinRatio, so that we can better observe the differences between co-scheduling heuristics. Results are quite similar to the results obtained with NPB-SYNTH.

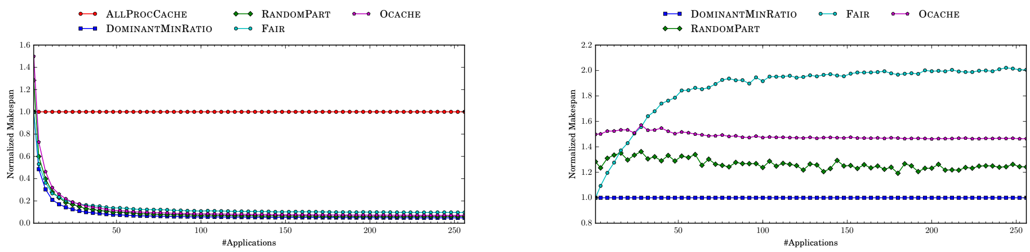

Figure 7: Impact of the number of applications with RANDOM.

A.2 Impact of the number of processors

Figure 8 (normalized with DominantMinRatio) shows the impact of the number of processors with 64 applications. Compared to Figure 5, the main difference is that Fair now obtains the worst performance, even 0cache is better. This difference in performance for Fair is due to a higher number of applications. As each application receive a fraction of cache and a fraction of processors, each of them obtains less resources when the number of applications increases.

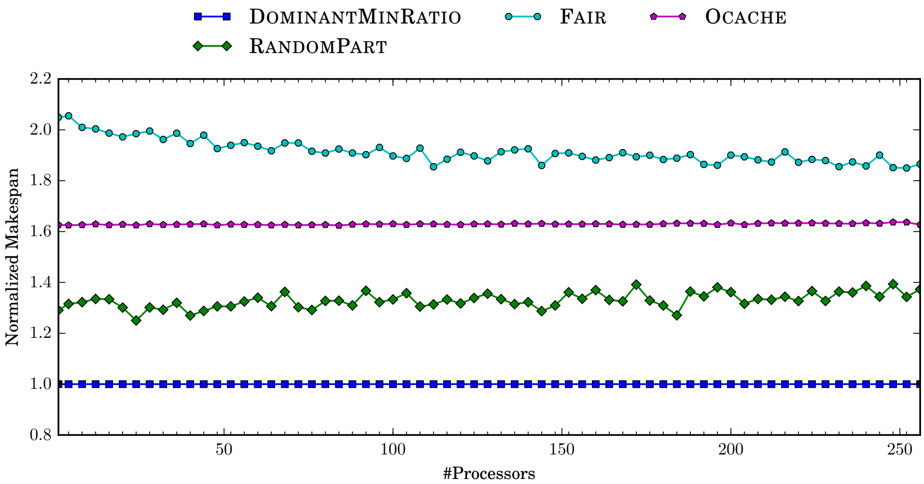

Figure 8: Impact of the number of processors with NPB-SYNTH and 64 applications.

Figure 9 (normalized with AllProcCache and DominantMinRatio) shows the impact of the number of processors with NPB-6. The number of applications is fixed to 6. We observe with less applications that Fair obtains better results than 0cache when the number of processors is bigger than 50.

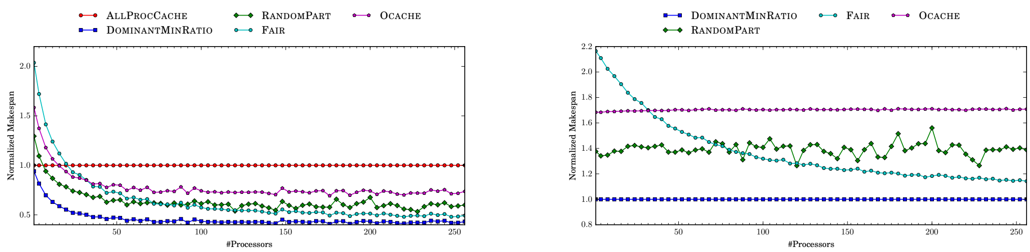

Figure 9: Impact of the number of processors with NPB-6.

Figure 10 (normalized with AllProcCache and DominantMinRatio) shows the impact of the number of processors with RANDOM. The number of applications is fixed to 16. We obtain similar results with RANDOM and NPB-SYNTH.

Figure 11 (normalized with AllProcCache and DominantMinRatio) shows the impact of the number of processors with RANDOM and 64 applications. As expected, we obtain similar

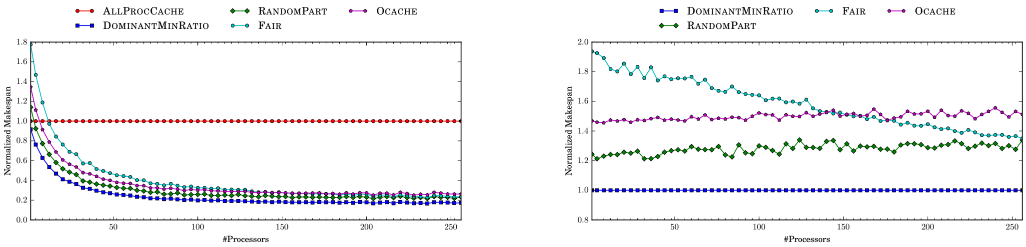

Figure 10: Impact of the number of processors with RANDOM and 16 applications.

results, 0cache and RandomPart show better performance when the number of applications increases. DominantMinRatio is still the best heuristic, the number of processors does not affect relative performance.

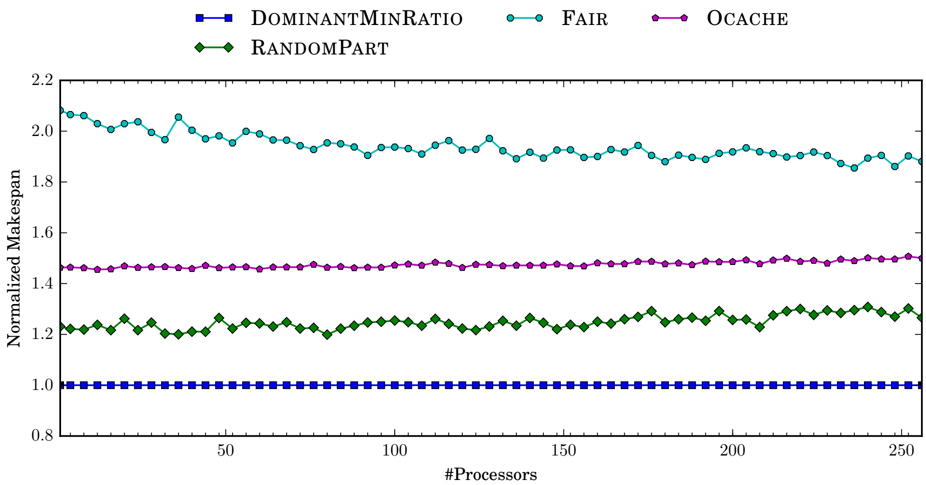

Figure 11: Impact of the number of processors with RANDOM and 64 applications (normalized with DominantMinRatio).

A.3 Impact of the sequential fraction of work

Figure 12 (normalized with AllProcCache and DominantMinRatio) shows the impact of the sequential fraction of work with NPB-6 and 6 applications. As in Section 6, results are also normalized with DominantMinRatio, in order to show the differences between heuristics. We observe that the performance of Fair increases when the sequential fraction of work increases. Indeed, more the sequential fraction of work is important, more the cache allocation becomes crucial.

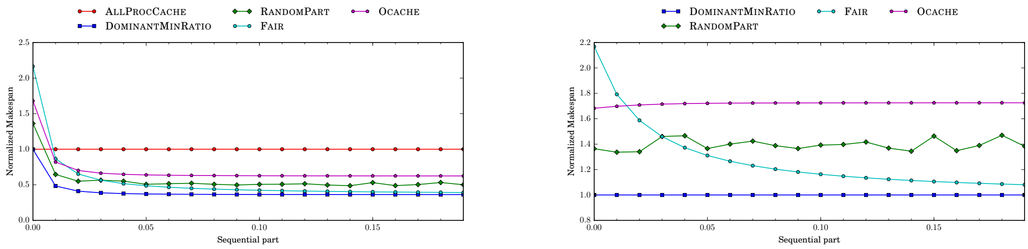

Figure 12: Impact of sequential fraction of work with NPB-6.

Figure 13 (normalized with AllProcCache and DominantMinRatio) shows the impact of the sequential fraction of work with RANDOM and 16 applications. We observe similar results to the previous one obtained with NPB-SYNTH.

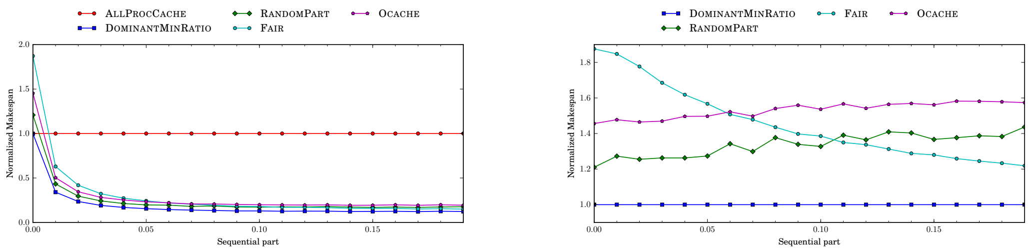

Figure 13: Impact of sequential fraction of work with RANDOM.

A.4 Impact of the cache latency

Figure 14 (normalized with AllProcCache) shows the impact of the cache latency l_s with NPB-SYNTH and 16 applications on 256 processors. The sequential fraction of work is fixed to $s_i = 0.0001$ for all i. We observe that the l_s cost does not have an impact on relative performance.

Figure 15 (normalized with AllProcCache) shows the impact of the cache latency l_s with NPB-SYNTH and 64 applications on 256 processors. The sequential fraction of work is fixed to $s_i = 0.0001$ for all i. As on the previous figure, we see that the l_s cost does not have an impact of relative performance, even with 64 applications.

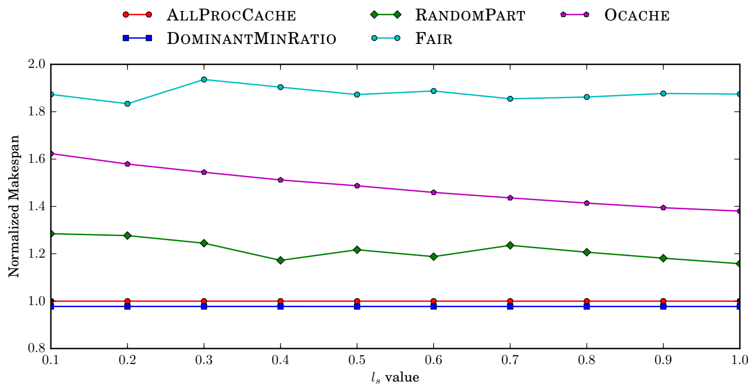

Figure 14: Impact of latency l_s with NPB-SYNTH and $s_i = 0.0001$ for all i.

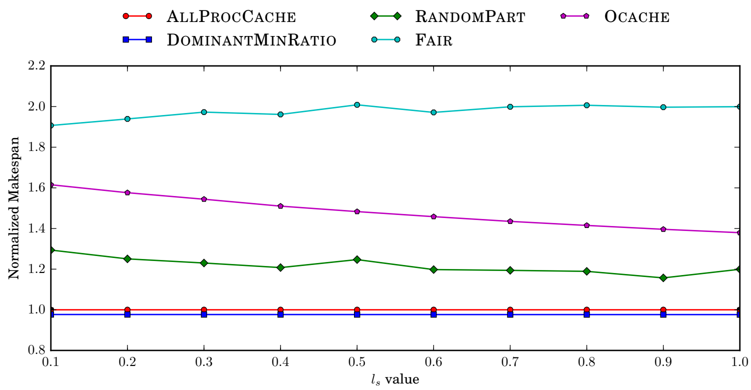

Figure 15: Impact of latency l_s with NPB-SYNTH and 64 applications.

A.5 Impact of the cache miss rate

Figure 16 (normalized with DominantMinRatio) shows the impact of the cache miss rate with NPB-SYNTH and 16 applications. We vary the cache miss rate $m^i_{40\mathrm{MB}\mathcal{S}_s}$ between 0 and 1. When the cache miss rate increases, the performance of RandomPart and 0cache increases. Indeed, when the rate of miss increases, using the cache is less important, so 0cache becomes competitive. But, we have to keep in mind that, with real applications, the cache miss rate rarely exceeds 20%.

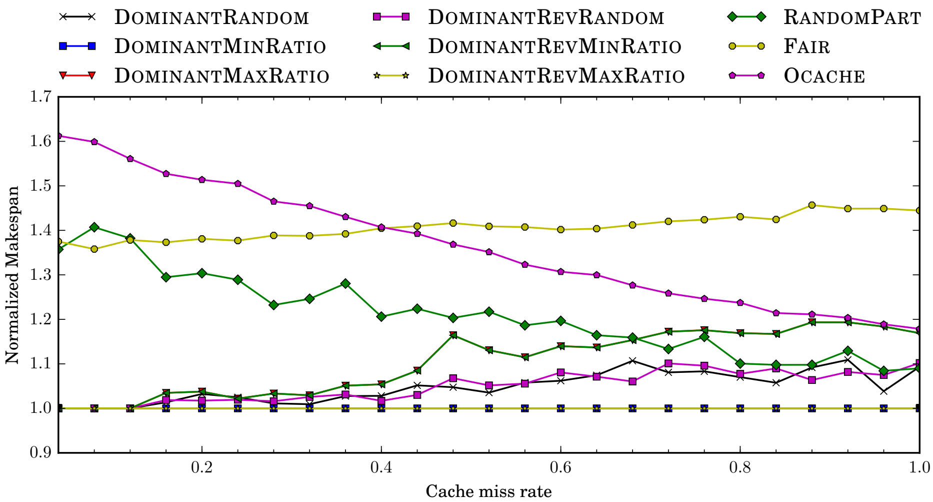

Figure 16: Impact of cache miss rate using a 1GB LLC.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399