PBBCache: an open-source parallel simulator for rapid prototyping and evaluation of cache partitioning and cache-clustering policies

Adrian Garcia-Garcia

Complutense University of Madrid
Facultad de Informática,
Calle Profesor García Santesmases 9, Madrid 28040, Spain
Phone: +34 (91) 3944394
Fax: +34 (91) 3944687

Juan Carlos Saez*

Complutense University of Madrid
Facultad de Informática,
Calle Profesor García Santesmases 9, Madrid 28040, Spain
Phone: +34 (91) 3944892
Fax: +34 (91) 3944687

José Luis Risco-Martin

Complutense University of Madrid
Facultad de Informática,
Calle Profesor García Santesmases 9, Madrid 28040, Spain
Phone: +34 (91) 3947543
Fax: +34 (91) 3944687

Manuel Prieto-Matias

Complutense University of Madrid
Facultad de Informática,
Calle Profesor García Santesmases 9, Madrid 28040, Spain
Phone: +34 (91) 3944550
Fax: +34 (91) 3944687

*This is the principal corresponding author

Email addresses: adriagar@ucm.es (Adrian Garcia-Garcia), jcsaezal@ucm.es (Juan Carlos Saez), jlrisco@ucm.es (José Luis Risco-Martin), mpmatias@ucm.es (Manuel Prieto-Matias)

Preprint submitted to Journal of Computational Science *March 6, 2020*

Abstract

Chip multicore processors (CMPs) constitute the architecture of choice for a wide spectrum of computing systems, ranging from power-efficient mobile devices to high-performance server platforms. Despite their benefits, the contention that appears when multiple applications compete for the use of shared resources among cores, such as the last-level cache (LLC), may lead to substantial performance degradation. This may have a negative impact on key system metrics such as throughput and fairness. Partitioning of the LLC (i.e., assigning a separate cache partition with a certain size to each application) has been proven effective to mitigate contention-related effects.

In this article we propose a parallel simulator that makes it possible to quickly compare the effectiveness of different cache-partitioning policies with the optimal solution for different optimization objectives. The simulator can obtain the optimal solution for any point during the execution of a multi-program workload where each application goes through a certain program phase. Our proposal leverages a slowdown-prediction model that accounts for degradation due to cache sharing and memory-bandwidth contention, which constitute the major factors of shared-resource contention on current CMPs. To determine the optimal solution for two optimization objectives (throughput and fairness optimization), we leverage a novel distributed-memory parallel branch-and-bound strategy specifically designed to efficiently distribute the computation across multiple processing cores.

Keywords: multicore processors, cache partitioning, branch and bound, simulation, HPC, Intel CAT, Python, ZeroMQ, ipyparallel

1. Introduction

Today, chip multicore processors (CMPs) constitute the architecture of choice for most modern general-purpose computing systems and will likely continue to be dominant in the near future. Despite its benefits, CMPs pose a number of challenges to the system software. One of the major challenges is how to mitigate the effects that come from contention on shared resources [1]. This contention stems from the fact that cores in a CMP are not truly independent processors but instead typically share a last-level cache (LLC) and other memory-related resources with the remaining cores, such as a DRAM controller and a memory bus or interconnection network [2, 3]. Applications

running simultaneously on the various cores may compete with each other for these shared resources, which could degrade their performance unevenly [4].

Partitioning of the shared LLC (i.e., assigning a separate cache partition with a certain size to each application in a workload) has been proven effective to mitigate shared resource contention effects [5, 6]. Recently, cache-partitioning hardware support has been adopted in commodity Intel processors via the Intel Cache Allocation Technology (CAT) [7] – part of Intel RDT. Because these extensions are available directly in privileged processor modes – where the operating system (OS) kernel or the virtual machine monitor (VMM) usually run – the system software constitutes the most natural place to implement cache-partitioning algorithms. At this level, we can efficiently make use of these hardware extensions (i.e. without costly system calls) along with hardware performance monitoring counters (PMCs), which provide valuable information on the cache-access behavior of applications, making it possible to guide cache-partitioning algorithms on-line [4, 8].

Recently, several cache-clustering algorithms [4, 8, 9] have been proposed. Cache clustering (aka partition-sharing) constitutes a generalization of strict cache partitioning, where, instead of assigning applications to separate cache partitions, each partition can be shared by a group of applications (aka *cluster*). Partitioning the cache optimally for a certain optimization objective is an NP-hard problem [6], but determining the optimal cache-clustering solution adds a new level of complexity, as a decision must be made on how to best group applications into clusters, and how to optimally distribute cache space across clusters. Previous work has pointed out that on systems supporting a reduced number of partitions or the creation of coarse-grained cache partitions only (i.e., in the order of megabytes) cache clustering proves more effective than strict cache partitioning as the number of applications increases [4, 8]. This is due to the finer grained distribution of the cache space that naturally results from sharing cache ways between applications.

A number of challenges arise in the design and implementation of effective cache-partitioning and clustering algorithms in the system software. Firstly, making decisions based exclusively on application cache behavior does not always provide the best results. The degree of bandwidth contention, which largely depends on how the cache is partitioned [1, 10], may substantially degrade the performance of individual applications. Failing to consider bandwidth-related performance degradation may backfire by reducing the benefits of cache partitioning [1, 2]. Unfortunately, efficiently determining the combined performance degradation that comes from cache

sharing and bandwidth contention constitutes a difficult problem [11, 12]. Secondly, implementing a cache-partitioning or clustering algorithm in the system software (i.e. OS kernel or VMM) requires substantial programming effort due to the difficulties associated with kernel-level development, such as the fact that implementation errors can bring down the entire system; testing any change in the algorithm's code may require to build, reinstall the OS kernel and reboot the machine; or the need of creating a very efficient implementation that ensures a low system latency and, is free of any floating-point operations, as these are largely impractical at this level [13].

In this article, we present PBBCache, an open-source [14] parallel simulator enabling to efficiently evaluate cache-partitioning and clustering algorithms. PBBCache relies on offline-collected application performance data (e.g., instructions per cycle, memory bandwidth consumption, etc.) to approximate the degree of throughput, fairness or other relevant metrics for a workload under a particular partitioning approach. Our simulator has been designed to achieve two main goals. First, it is meant to serve as a tool for rapid prototyping and evaluation of partitioning policies. To increase programming productivity, PBBCache has been implemented in Python, which is one of the most widely used programming languages today [15]. The simulator allows researchers to guide the design process of their algorithms and, more importantly, to easily discard unpromising approaches without having to go through the tedious development process in the system software. The recent LFOC policy [8] is a clear example of effective partitioning scheme whose design process was guided with PBBCache. Second, the simulator has been built to enable the assessment of the real potential of partitioning algorithms and to identify their limitations, by providing a comparison with the optimal solution. Even though many cache-partitioning and clustering approaches have been proposed [4, 5, 9, 10], it still remains unclear how close they perform relative to the optimal solution. To fill this gap, PBBCache has the ability to efficiently determine the optimal solution for different optimization objectives using a distributed-memory parallel strategy.

Apart from the proposed simulator, we make the following contributions:

- PBBCache is equipped with a slowdown-prediction model enabling to determine the performance degradation that an application suffers due to cache-sharing and memory-bandwidth contention. To approximate bandwidth contention for a certain distribution of cache space across applications in a workload, we extended the probabilistic model pro-

posed in [11] with information on how sensitive an application is to a reduction in its effective bandwidth consumption at runtime.

- To efficiently determine the optimal cache space distribution, the simulator leverages a novel parallel distributed-memory branch-and-bound (B&B) strategy that follows the master-slave pattern. Note that this strategy has been specifically designed for the optimization problems that arise in the context of cache partitioning, and enables to effectively distribute the computation across cores on one or multiple computing nodes. A key design aspect is the mechanism used to break down the work to be done in parallel into tasks (referred to as *subnodes*) with a similar computational complexity, which provides good scalability. The effectiveness of the bounding functions we devised for various optimization objectives, also contributes to the success of the B&B approach.
- To the best of our knowledge, our proposal is the first parallel approach to solve the optimal cache-partitioning problem by factoring in both cache-sharing and memory-bandwidth contention. Specifically, we studied two optimization objectives – fairness and throughput optimization –, and found that each associated optimization problem can be expressed as a mixed-integer non-linear program. Notably, state-of-the-art non-linear solvers [16, 17, 18, 19] fail to provide a solution for these problems (encoded in AMPL), unlike our parallel B&B approach.
- To evaluate the effectiveness of PBBCache we implemented existing partitioning policies [5, 8, 9, 10] on top of it, and compared the results it provides with the actual figures observed on real hardware equipped with Intel-CAT enabled processors. Moreover, to assess the performance and scalability of the parallel B&B algorithm we conducted experiments using single-node and multi-node machine configurations.

The remainder of the paper is organized as follows. Sec. 2 presents background on cache partitioning. Sec. 3 discusses related work. Sec. 4 introduces PBBCache's design and its inner workings. Sec. 5 showcases our strategy to determine the optimal cache-partitioning solution via parallel B&B. Sec. 6 covers the experimental evaluation, and Sec. 7 concludes the paper.

2. Background

In this section we first describe the metrics we considered to assess different optimization objectives. Then, we formally introduce the problems of optimal cache partitioning and optimal cache clustering.

2.1. Metrics

To measure the performance degradation of an individual application a_i in a multi-program workload consisting of N applications $A = \{a_1, a_2, \cdots, a_N\}$ 125 we consider the *Slowdown* metric, defined as follows:

$$Slowdown_{a_i} = \frac{CT_{part,a_i}}{CT_{alone,a_i}} \tag{1}$$

where CT_{part,a_i} denotes the completion time of application a_i under a given cache-partitioning scheme and running together with the rest of applications in A; and CT_{alone,a_i} is the completion time observed for the application when it runs alone on the CMP system.

130 Defining the slowdown in terms of completion time, as in Eq. 1, allows us to calculate it for both multithreaded and single-threaded applications. In addition, for single-threaded (ST) applications the slowdown can be also expressed in terms of the average number of instructions per cycle observed when it runs alone (IPC_{alone,a_i}) and that achieved when it runs with other 135 applications in the workload (IPC_{part,a_i}), as follows:

$$Slowdown_ST_{a_i} = \frac{IPC_{alone,a_i}}{IPC_{part,a_i}} = \frac{CT_{part,a_i}}{CT_{alone,a_i}} \tag{2}$$

Previous research on fairness for CMPs [2, 20] define a scheme as fair if equal-priority applications in a workload A suffer the same slowdown as a result of sharing the system. To cope with this notion of fairness, we employ the *unfairness* metric [2, 3, 20] (lower-is-better), which is defined as follows:

$$Unfairness = \frac{MAX(Slowdown_{a_1},...,Slowdown_{a_n})}{MIN(Slowdown_{a_1},...,Slowdown_{a_1})} \tag{3}$$

140 To quantify throughput, previous works [4, 21] have employed the *System ThroughPut* (STP), which is defined as follows:

$$STP = \sum_{i=1}^{n} \left(\frac{CT_{alone,a_i}}{CT_{part,i}} \right) = \sum_{i=1}^{n} \left(\frac{1}{Slowdown_{a_i}} \right) \tag{4}$$

2.2. Optimal cache-partitioning problem

Before presenting a formal problem definition, it is worth describing how current cache-partitioning algorithms typically operate. For simplicity in

145 the explanation we will focus on way-partitioning, since this is the specific hardware implementation found in our experimental platforms (Intel CAT). Essentially, a partitioning algorithm has to distribute the available cache ways among applications based on their runtime properties so as to accomplish the objective it was designed to achieve (e.g., maximizing throughput,
150 minimizing energy consumption, etc.). Notably, this kind of algorithms, as well as the corresponding optimization problem we describe next, just determines the number of ways that will be allotted to each applications, but not which exact ways are assigned. An important challenge is that certain applications may go through different program phases, which may lead to
155 time-changing cache behavior. Under these circumstances, partitioning the cache statically (i.e. fixed cache way distribution throughout the execution) may not constitute the best solution. To cope with phase changes, partitioning algorithms are usually invoked periodically [4, 5, 8, 9]. Specifically, the system software continuously monitors application behavior by using (for
160 example) performance counters; the more recent values of the gathered performance metrics – these usually differ across algorithms – are used as input to the partitioning algorithm (invoked every so often) so as to determine the partitioning for the next execution interval, where the applications are likely to exhibit a similar behavior to that reflected by the recent collected data.

165 Our goal is to determine the optimal cache partitioning for a workload in a certain execution interval where the applications exhibit a stable behavior (no distinct program phases[1]). In providing researchers with this optimal solution generated off-line with our simulator for a certain optimization objective, they can quickly compare it against the solution provided by any
170 partitioning algorithm, which also makes the same assumption: applications will likely exhibit a stable behavior in the near future, which is similar to that reflected by the latest data collected. Making a comparison with multiple application mixes enables to quickly assess the real effectiveness of a partitioning algorithm. More importantly, this allows to identify potentially
175 conflicting workload scenarios where the algorithm fails to achieve good results, thus providing valuable insights to guide the algorithm's design process.

The optimal cache-partitioning problem can be generically formulated

[1] Determining an optimal solution for a workload considering program phases is a much more complex problem, especially because phase transitions do not happen at the same time in multiple applications. That would require to break down the whole workload execution into stages of stable behavior across applications, and apply our proposed method to detect the optimal for solution for each and every "stable stage".

as a Mixed Integer Program (MIP). Let A be a workload consisting of N applications $\{a_1, a_2, \cdots, a_N\}$ that run on a system featuring a W-way last-level cache with $W \geq N$, and let K be $\{1..W\}$. The set DV of decision variables is defined as $\{w_{a,k} \mid \forall a \in A, k \in K\}$, where each decision variable $w_{a,k}$ is a binary variable indicating whether or not an application $a \in A$ is assigned k ways. The associated MIP is formulated by considering a generic optimization function f, as follows:

$$\textit{Minimize: } f(DV) \tag{5}$$

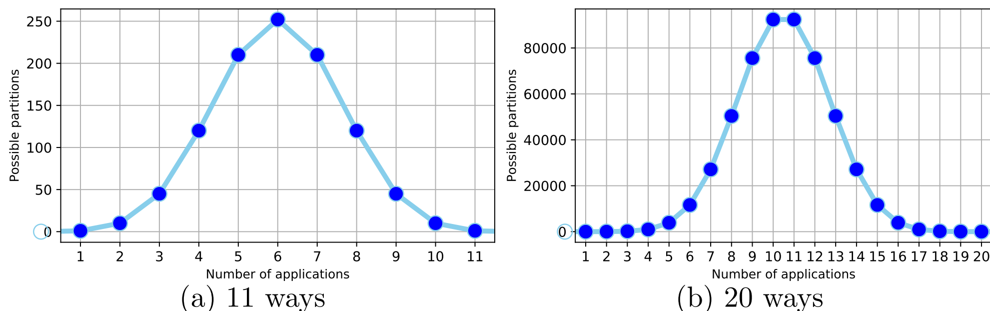
Subject to:

$$\sum_{k \in K} w_{a,k} = 1, \forall a \in A \quad \textit{Only 1 way assignment per application} \tag{6}$$

$$\sum_{a \in A} \sum_{k \in K} k \cdot w_{a,k} = W \quad \textit{No cache ways remain unused} \tag{7}$$

$$1 \leq \sum_{k \in K} k \cdot w_{a,k} \leq W - N + 1, \forall a \in A \quad \textit{Each application gets at least 1 way} \tag{8}$$

In this work we focus on two specific optimization problems, whose definition entail incorporating non-linear constraints to the set of Eqs. 5 to 8. The first problem, referred to as *Opt-STP*, is system throughput optimization, and it comes down to maximizing the STP metric. The second one, denoted as *Opt-Unf*, strives to find the cache space distribution that minimizes the Unfairness metric, so as to optimize system-wide fairness. Both the STP and Unfairness metrics depend on the slowdown experienced by each application, which in turn depend on the cache-way distribution (decision variables in the generic MIP). Notably, *Opt-STP* and *Opt-Unf* constitute non-linear optimization problems. This stems from the fact that evaluating either optimization function (STP or Unfairness) for any feasible cache-way distribution requires determining the slowdown of each application by means of a prediction model that factors in the combined performance degradation due to cache and bandwidth contention. In our proposed model, this entails solving a set of non-linear equations, as we will describe in Sec. 4.2. The detailed formalization of *Opt-STP* and *Opt-Unf* as Mixed Integer Non-Linear Problems (MINLPs) can be found in the Appendix. Notably, we created AMPL implementations for these problems and tested them with different state-of-the-art non-linear solvers [16, 17, 18, 19], but found that all of them failed to provide a solution. More importantly, even for small workloads – where PBBCache finds the optimal solution sequentially in less than one



(a) 11 ways (b) 20 ways

Figure 1: Number of possible ways to partition a LLC as we increase the number of applications for 11 and 20 cache ways, respectively. The number of ways and applications considered are based on the features of Platforms A and B, described in Sec. 6.1

minute – they also failed to find a local minima after 1 hour of execution. We also tried feeding the solvers with an initial solution determined earlier via a heuristic algorithm, which was unable to provide the optimal solution for the workloads considered. In this case, the solvers were not even able to find a better solution in one hour of execution (we configured the solvers so that the execution was aborted automatically if a near-optimal solution was not found within that time period). By contrast, for the largest workloads we tested with (see Sec. 6), our simulator is capable of finding an optimal solution in less than 9 minutes by using via a sequential algorithm, and in less than 34s by leveraging a parallel B&B strategy on a 28-core server platform.

For the sake of completeness, the following recursive definition provides the number of possible ways P to partition a W-way LLC for N applications:

$$P(W,N) = \begin{cases} 1 & W = N \text{ or } N = 1 \\ N & W = N+1 \\ \sum_{i=1}^{W-N+1} P(W-i, N-1) & otherwise \end{cases} \tag{9}$$

As shown in Fig. 1, the P function reaches the maximum when $N = \lceil W/2 \rceil$. The number of possible ways to partition the LLC rapidly increases with the application count (N), but it drops back symmetrically towards 1 when $N > \lceil W/2 \rceil$. Due to the vast search space when W is high, determining the best solution via extensive exploration is largely impractical.

2.3. Optimal cache-clustering problem

Optimal cache clustering constitutes a generalization of the optimal cache-partitioning problem. When cache clustering is used, applications in the workload are grouped into a number of sets, each one referred to as a *clus-*

ter; the cache space is divided into separate partitions, one for each cluster. So, applications in the same cluster share the same cache partition.

To formally define the optimal cache clustering for a certain workload A consisting of N applications, we first introduce some basic terminology. We use the term *cluster set* to refer to any possible way to break down A into *clusters* so that the available cache space (W ways) can then be distributed across clusters (i.e. each cluster gets at least one way). A cluster set CS is one of the possible partitions of the A set (i.e. grouping of the set's elements into non-empty subsets) with a number of items $\leq W$. Let $\mathbb{C}_A$ be the set with all possible cluster sets of A. Note that $|\mathbb{C}_A| \leq \mathcal{B}_N$, where $\mathcal{B}_N$ denotes the Bell number, namely the number of possible partitions of the A set. Let f be the objective function to be minimized in the optimization problem. We define $OPT_{CS,f}$ for a certain cluster set $CS \in \mathbb{C}_A$, as the value of f associated with the distribution of cache ways across clusters in CS that minimizes f.

For workload A and a certain optimization function f, we define the optimal cache clustering as the cluster set in $\mathbb{C}_A$ that exhibits the optimal (minimal) $OPT_{CS,f}$ value. A potential way to determine the solution to the optimal cache-clustering problem is to generate $\mathbb{C}_A$ and then identify the optimal solution by comparing the $OPT_{CS_i,f}$ values for each $CS_i \in \mathbb{C}_A$. In turn, determining $OPT_{CS_i,f}$ for any CS_i constitutes an instance of the optimal cache-partitioning problem, where cache space is distributed among clusters rather than among applications. Given the non-linear nature of Opt-STP and Opt-Unf, determining the optimal cache clustering for the throughput and fairness optimization objectives is also a non-linear problem.

3. Related Work

Our PBBCache simulator is a tool for rapid prototyping and evaluation of cache-partitioning and cache-clustering approaches, and it is equipped with several sequential and parallel B&B algorithms to solve the Opt-STP and Opt-Unf optimization problems. In discussing related work we first consider partitioning policies. Next, we cover previous research on parallel B&B.

3.1. Cache-partitioning and cache-clustering policies

A large body of work has studied the cache-partitioning problem and proposed different approaches to determine promising solutions using approximate algorithms [5, 9, 10, 22]. A recent survey [6] discusses the most effective approaches for various optimization objectives, such as maximizing throughput or reducing energy consumption. Specifically, in our simulator

we implemented the UCP [5] and Yu-Petrov [10] algorithms – described in Sec. 4.4, which primarily strive to optimize system throughput.

More recently, different cache-clustering algorithms have been proposed [4, 8, 9]. We implemented the KPart [9] and LFOC [8] policies in PBBCache. Both policies are described in detail in Sec. 4.4. LFOC was proposed as part of our earlier work [8], and constitutes the first partitioning algorithm whose design process was guided with PBBCache. Specifically, LFOC tries to mimic the behavior of the optimal clustering solution on platforms featuring an Intel Skylake processor, which we were able to approximate with our simulator. Other authors [4] have proposed partitioning algorithms where partitions overlap with each other; these schemes do not constitute pure cache-clustering approaches according to the definition presented in Sec. 2. The LFOC approach is able to deliver better performance and fairness than this kind of algorithms, as shown in [8].

We should highlight that none of these works [4, 5, 9, 10] provide a explicit comparison of the corresponding proposal with the optimal cache-partitioning solution, as we do in this work. Hence, this analysis constitutes an important contribution of our article.

3.2. Parallel Branch-and-Bound

The B&B method constitutes a classical approach to solve combinatorial optimization problems. It can be considered a search space enumeration that explores a subset of feasible solutions. The effectiveness of an implementation of the B&B method for a specific problem largely depends on several design aspects, such as the bounding function used for pruning, the rule to select the next node to be processed, or the mechanism to determine the initial solution [23]. In minimization problems, the bounding function returns a *lower bound* of the cost (i.e. value of the optimization function used) of the best solution reachable from a specific node. B&B algorithms for minimization problems maintain a variable with the cost of the best solution found thus far (aka incumbent), which is an *upper bound* of the cost of the optimal solution. Conversely, in maximization problems, the bounding function returns an *upper bound* of the cost of the best solution reachable from a node, and a variable is used to maintain a *lower bound* of the cost of the optimal solution.

As an illustrative example, Algorithm 1 depicts the pseudo-code of a sequential implementation of the B&B method for a minimization problem. A heuristic algorithm is used to determine the initial solution, which is used for the initialization of the *upper bound* (lines 1-3). A priority queue is

Algorithm 1: Sequential B&B algorithm. (variant of the one defined in [24]).

```
1  incumbent ← initial solution provided by a heuristic algorithm
2  upper_bound ← incumbent.Cost();
3  prio_q ← [root node];
4  while prio_q is not empty do
5      node ← pop highest priority node from prio_q
6      foreach child of node do
7          if child.isSolution() && child.Cost() < upper_bound then
8              (incumbent, upper_bound) ← (child, child.Cost());
9              Remove nodes from prio_q whose cost > upper_bound;
10         else
11             lower_bound ← bounding_function (child);
12             child.setLowerBound(lower_bound);
13             if child is a feasible node && lower_bound < upper_bound then
14                 prio_q.append(child);
15             end
16         end
17     end
18 end
```

used to keep nodes as they are expanded during the search (line 14). As for the node selection rule, we observed that using best-first search in the optimal cache-partitioning problem provides better performance than depth-first search. In addition, the former approach is less sensitive to the order in which applications are listed in the workload (that determines the area of the search-space tree where the optimal solution is located). Specifically, the node in the queue with the smallest lower bound is processed first.

An important contribution of this work is the parallel B&B strategy that our simulator leverages to determine the optimal cache-partitioning solution. As we explain in Sec. 5, it is a distributed-memory strategy that follows a master-slave pattern. The parallelization of B&B has been widely studied since it represents a classical high-level problem-solver paradigm in computer science [23, 24, 25]. There are critical implementation challenges that must be faced [24, 25], such as the following: initial definition of the search space, work allocation policies, communication of key information between processes, the minimization of idleness and the maximization of useful work. Our parallel strategy has been carefully designed to cope with many of these challenges that also arise in context of the Opt-STP and Opt-Unf non-linear problems. To the best of our knowledge, ours constitutes the first attempt to efficiently solve the optimal cache-partitioning problem via parallel B&B when factoring in both cache-sharing and memory-bandwidth contention.

Crainic et al. [24] group parallel B&B algorithms into two main categories: *tree-based* and *node-based* strategies. Algorithms in the first category aim to build and explore the search space tree in parallel. By contrast, *node-based*

approaches aim to accelerate a particular operation mainly at the node level, such as evaluation or bounding. The vast majority of the proposed parallel B&B algorithms exploit a *tree-based* strategy or a combination of node and tree parallelization [26, 27, 28, 29, 30]. Many frameworks have also been proposed to simplify the development of parallel B&B algorithms [24]. A well-known example is Bobpp [28, 30, 31], which allows the implementation of combinatorial problem solvers on both shared and distributed-memory architectures. Notably, the use of a framework is not always the best approach when dealing with a specific problem since custom solutions can take into account some characteristics that optimize performance [28]. Indeed, our implementations feature specific optimizations tailored to the Opt-Unf and Opt-STP problems. A key aspect of our approach is the fact that it considers *subnodes* as the work unit. As we explain in Sec. 5.3, the subnode abstraction allows us to break down the associated processing of a single node into tasks with similar computational complexity, which enables to deliver better scalability than that achieved by considering coarser-grained work units. This one and many other design aspects of our strategy are motivated by the high computational cost associated with evaluating the bounding and optimization functions, which require solving a set of non-linear equations.

4. Design of the PBBCache simulator

In this section we provide an overview of the simulator design. We begin by introducing the structure of the simulator's input data and the interaction with it via the command line. Then, we describe the technique used to approximate the slowdown for each application in a workload based on the amount of cache space allotted and the degree of memory-bandwidth contention. Finally, we outline the partitioning algorithms implemented in our cache-partitioning simulator, and showcase some implementation details.

4.1. Input data and command-line options

PBBCache is a command-line tool. As depicted in Fig. 2 it accepts as input two text files: a *workloads* file and a *metrics* file. The *workloads* file specifies the composition of the workloads that will be used in the simulation; each workload (one per line) is encoded as a comma-separated list of application names. The *metrics* file stores a table, where each row contains the values of various runtime metrics (e.g. instructions per cycle, cache miss rates on different cache levels, memory bandwidth consumption, memory-related

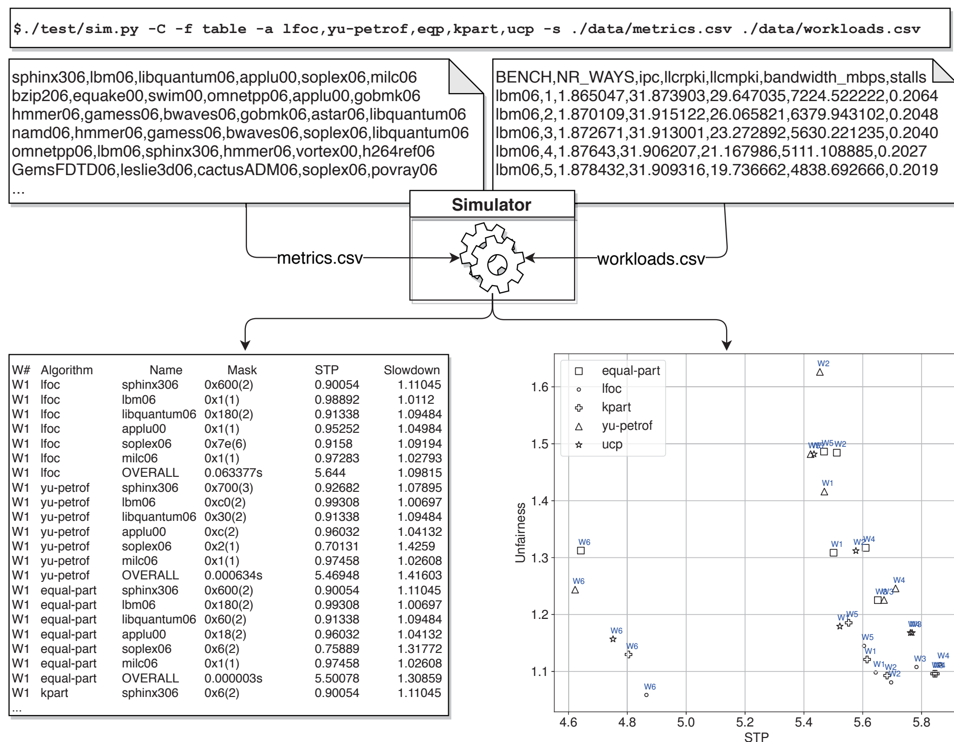


Figure 2: Simulator's diagram that shows an example of the user interaction with the simulator via command line, the data input and the generated output.

stall cycles, etc.) for a specific application, which have been gathered offline with PMCs when the application runs alone on a certain platform with a fixed number of cache ways. Essentially, this file summarizes the behavior of each application with every possible cache way count. The various metrics, which can be easily gathered on Intel processors that support Intel CAT, are used as input to different partitioning algorithms (as discussed in Sec. 4.4 each algorithm uses different metrics), and are also required to determine both the slowdown (see Sec. 4.2) and the amount of cache space each application gets inside a cluster (see Sec. 4.3). In creating the *metrics* file, the user may decide to include the information only for a particular program stage (e.g. first K billion instructions, as in [8]), a specific execution phase or the average registered for each metric throughout the application's execution.

The simulator text output, which can be presented in different formats (`-f` option), shows the amount of cache ways allotted by each partitioning algorithm considered for the simulation (as indicated with `-a`) as well as other values, such as the per-application slowdown. In using the sample command

370 of Fig. 2, where the `-C` option is used, a chart will be also generated making it possible to quickly compare the effectiveness of the various algorithms regarding fairness and throughput[2]. Simulations are performed sequentially by default, but because they are independent from one another they can be also launched in parallel (when the `-P` option is provided) by leveraging
375 multiple slave processes running on one or multiple machines; details on our parallel programming framework can be found in Sec. 4.5. A complete discussion of PBBCache's command-line options can be found in [14].

4.2. Determining the slowdown under cache-partitioning

For each workload and partitioning algorithm indicated by the user in the
380 command line, PBBCache determines the slowdown of each application in a workload, which is necessary to assess the degree of fairness and throughput delivered. Each partitioning algorithm decides how the various LLC cache ways are distributed among applications in a workload. The cache space distribution has an important impact on performance, but also determines
385 the level of bandwidth contention present on the system, which may lead to performance degradation. Therefore, to accurately determine the slowdown of each application both the allotted cache ways and the degree of bandwidth contention should be considered. Specifically, let A be a workload consisting of N applications $[a_1, \cdots, a_N]$ running on a system that features a W-way
390 last-level cache, and under a certain cache-partitioning algorithm *part*. Our simulator approximates the slowdown of each application a_i as follows:

$$Slowdown_{a_i} = SC_{\mathrm{part,a}_i} \cdot SB_{\mathrm{part,a}_i} \quad (10)$$

where $SC_{\mathrm{part,a}_i}$ indicates how much the application slows down due to the amount of cache space granted by *part* to it (w_i ways); $SB_{\mathrm{part,a}_i}$ is the slowdown that a_i suffers exclusively due to bandwidth contention (see Sec. 4.2.1).
395 PBBCache approximates $SC_{\mathrm{part,a_i}}$ with the ratio of instructions per cycle (IPC) observed for a_i when using W and w_i ways:

$$SC_{\mathrm{part,a}_i} = \frac{\mathrm{IPC}_{a_i}(W)}{\mathrm{IPC}_{a_i}(w_i)} \quad (11)$$

Determining $SB_{\mathrm{part,a}_i}$ is a more challenging task for two reasons. First,

[2]Because the value of the unfairness metric only depends on the maximum and minimum slowdown observed across applications, reporting the value of the STP metric as well is crucial to properly assess the effectiveness of a partitioning approach [8].

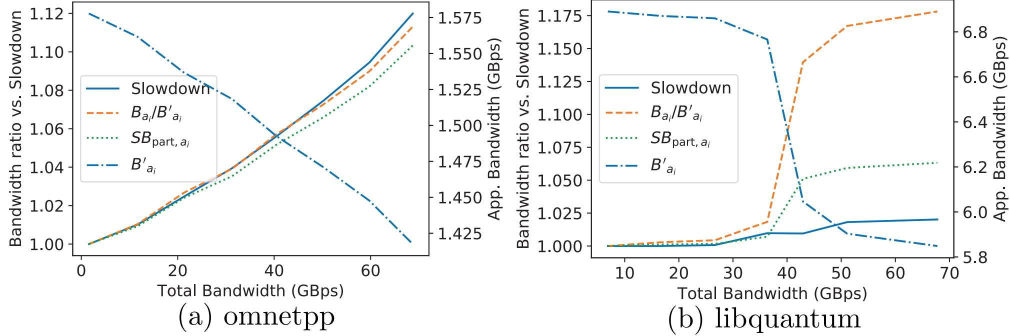
the amount of memory bandwidth consumed by a_i at runtime depends on w_i and on the bandwidth consumption of the remaining programs [11]. So, the
400 behavior of each application under the cache-way distribution made by *part* has to be taken into consideration to determine $SB_{\mathrm{part,a}_i}$. Second, we found that this performance degradation (slowdown factor) depends on how sensitive the application is to bandwidth contention. We now proceed to describe the bandwidth model that PBBCache leverages to approximate $SB_{\mathrm{part,a}_i}$.

405 *4.2.1. Modeling Memory Bandwidth Contention*

To illustrate the effects of bandwidth contention on our experimental platforms, we conducted several experiments where an application runs simultaneously with an increasing number of aggressor benchmarks (i.e., a bandwidth-intensive synthetic benchmark [32]). For each application we
410 measured how its bandwidth and slowdown – w.r.t. the solo execution – varies as we add more instances of the aggressor application (increasing the total bandwidth consumption of the workload). To effectively track the slowdown that comes primarily from memory bandwidth contention in the experiments, we assigned separate cache partitions for the application under
415 study and for the aggressors. Fig. 3 shows the results for two SPEC CPU2006 applications gathered on Platform B. (more details on that platform, which features a 20-core Intel Xeon Skylake processor, can be found in Sec. 6.1.). Note that B_{a_i} denotes the bandwidth consumption of the application when it runs alone on the platform (constant), and B'_{a_i} represents its actual band-
420 width when running with the remaining applications in the workload.

As shown in Fig. 3, B' drops as we increase the total bandwidth consumption, whereas the slowdown increases. Clearly, the observed slowdown is not negligible (e.g., up to 1.12x for `omnetpp`), so bandwidth-contention related degradation must be factored in to accurately determine the slowdown for
425 individual applications. Notably, on Platform A, where we also conducted the same experiments we observed considerably higher slowdowns due to bandwidth contention (up to 1.7x). As pointed out in Sec. 6.2, Platform A has roughly half the available bandwidth of Platform B, hence the larger observed slowdowns.

430 To properly account for bandwidth contention effects, PBBCache employs a variant of the probabilistic model proposed by Morad et al. [11]. Essentially, this model enables to approximate – from offline-collected information of individual applications running alone – (i) the bandwidth that each application would exhibit when running simultaneously with others and (ii) the



(a) omnetpp (b) libquantum

Figure 3: Memory bandwidth vs. slowdown observed for `omnetpp` and `libquantum` as increasing the total memory bandwidth consumption. The slowdown prediction provided by Morad's model (B_{a_i}/B'_{a_i}) and PBBCache's model (SB_{part,a_i}) is also reported.

slowdown that comes exclusively from memory-bandwidth contention. Essentially to determine (i) for a workload consisting of N applications, the following system of $N+1$ non-linear equations must be solved:

$$\left\{ \overline{B'}^{2}_{a_i} \cdot \left(1 - \frac{1}{\overline{B}_{a_i}}\right) + \overline{B'}_{a_i} \cdot \left(1 - \frac{1}{\overline{T}}\right) \cdot \left(1 - \frac{1}{\overline{B}_{a_i}}\right) + 1 - \frac{1}{\overline{T}} = 0 \right\}_{i=1}^{N} \tag{12}$$

$$\textstyle\sum_{i=1}^{N} \overline{B'}_{a_i} = \overline{T} \tag{13}$$

where $\overline{B}_{a_i}$ is the bandwidth observed for each application a_i when running alone on the platform (with the same amount of cache space as that alloted in the workload), $\overline{B'}_{a_i}$ is the actual bandwidth for a_i when running simultaneously with the other applications in the workload, and $\overline{T}$ is the total bandwidth consumption of the workload. Note that $\overline{B}_{a_i}$, $\overline{B}'_{a,i}$ and $\overline{T}$ are normalized to the maximum memory bandwidth of the platform.

To determine the application slowdown due to bandwidth contention (ii) Morad's original model uses the ratio $\overline{B}_{a_i}/\overline{B'}_{a,i}$, which is based on the observation that the application bandwidth consumption and its performance naturally decreases due to contention, and so does its performance. We observed that this approach to approximate the slowdown is accurate for highly bandwidth-intensive applications (i.e., over 90% of its pipeline stall cycles are dominated by long-latency demand cache misses) such as `omnetpp` (see Fig. 3a). However, for the remaining applications, the reduction in memory bandwidth consumption does not correlate linearly with the performance degradation, thus obtaining inaccurate slowdown estimates with the model (such as on Fig. 3b). Essentially, some applications can generate a lot of prefetching-related memory requests and cache writebacks operations, which

may result in high memory bandwidth consumption; however, a reduction of the bandwidth consumption of these applications due to contention does not directly translate into linear performance degradation. To mitigate this issue in calculating the slowdown of an application a_i, PBBCache factors in the stall cycles due to long-latency demand cache cycles (MS_{a_i}) and the total number of stall cycles (TS_{a_i}) observed in the solo execution as follows:

$$SB_{part,a_i} = \frac{TS_{a_i} + MS_{a_i} \cdot \left(\frac{\overline{B}_{a_i}}{\overline{B'}_{a_i}} - 1 \right)}{TS_{a_i}} \quad (14)$$

4.3. Determining the slowdown for cache-clustering policies

As explained in Sec. 2.3, cache-clustering policies group applications into clusters; each cluster is assigned a separate cache partition with a certain size. To apply the slowdown prediction model presented in Sec. 4.2 (Eq. 10-14), the simulator must determine first how much cache space each application gets inside the assigned cluster. This is a challenging task, as the effective cache space an application gets largely depends on its co-runners in the cluster [1].

Caches in modern processors typically implement a variant of the pseudo-LRU replacement policy [1, 33][3]. Under these circumstances, the amount of cache space that an application gets is usually proportional to its rate of demand (frequency of cache misses) in competition with the rate of demand of the co-runners [5]. Based on this observation Mukkara et al. [22] proposed a simple model to rapidly estimate the effect in the cache miss rate curves when several applications share a cache, and to approximate the cache space that each application would get for different way counts. The KPart clustering policy relies on this model [9], which leverages per-application MPKI (cache Misses Per Kilo-Instruction) tables. Essentially, for each number of cache ways the model determines the fraction of that cache space that will be assigned to each application. Intuitively, the higher the MPKI of an application for a certain way count, the more space the application gets when sharing a portion of cache with that number of ways.

By comparing the prediction provided by Mukkara's model with the actual cache usage reported by the Intel Cache Monitoring Technology on our experimental platforms, we observed that using the MPKI for the cache space prediction may lead to substantial inaccuracies that stem from the fact that

[3]The last-level cache, however, may incorporate specific optimizations to increase effective associativity without adding ways [33].

the MPKI is not a good proxy of the rate of cache demand. Specifically, two applications with the same MPKI value but different performance (IPC) have a different rate of cache demand (in terms of misses per cycle). The application with the higher IPC in this case has a higher rate of demand, and so it typically obtains more cache space.

To overcome this shortcoming, our simulator employs a variant of Mukkara's model that uses MPKC (Misses Per Kilo Cycles) tables instead of MPKI tables. We refer to this model as the *cache-space* model. Note that the predicted amount of cache space for an application in a cluster may not be a multiple of the way size (e.g. 1.5 ways). In this case, linear interpolation (as in [9, 22]) is used to determine the value of the different metrics (i.e. IPC_{a_i}, $\overline{B}_{a_i}$, TS_{a_i} and MS_{a_i}) required to apply our slowdown-prediction model. So for example, if an application is expected to receive 1.5 ways of cache space inside a certain cluster, the value of each metric would be obtained via linear interpolation based on the corresponding metric values for 1 and 2 ways, which can be found in the *metrics* file.

4.4. Partitioning policies

The current version of the simulator implements five cache-partitioning schemes: Equal-Part, UCP [5], Yu-Petrov [10], KPart [9] and LFOC [8]. Our simulator employs the offline-collected metrics found in the *metrics* file as input to each partitioning algorithm. Note that real implementations in the system software of most of these (approximate) algorithms may gather runtime application metrics online using PMCs.

The **Equal-Part** approach is a naive approach, used for comparison purposes, that assigns all applications a separate partition with the same size.

UCP aims at improving fairness and overall throughput by minimizing the total number of misses incurred by all applications in the workload on the shared last-level cache. UCP does not attempt to determine the optimal solution but instead employs an approximate algorithm referred to as *lookahead* [5], which uses as input the MPKI table of each application. This table stores the application's MPKI value for any possible cache size.

The algorithm proposed by **Yu and Petrov** [10] strives to reduce system bandwidth pressure. To this end, it partitions the LLC so as to minimize the total bandwidth. The algorithm relies on per-application bandwidth consumption measurements with different cache sizes gathered offline.

KPart [9] constitutes a cache-clustering approach designed for throughput optimization. KPart implements an iterative algorithm that creates and

525 merges application clusters via hierarchical clustering. To decide which clusters must be merged on each iteration of the loop and how to distribute the available ways among clusters (inter-cluster way-partitioning), the scheme leverages the distance metric proposed in [22] as well as the UCP approach [5]. The application of UCP and the evaluation of the distance metric relies on
530 the ability to determine MPKI tables and IPC tables (i.e. number of Instructions Per Cycle for different cache sizes) online for each application.

LFOC [8] is a lightweight cache-clustering policy that seeks to enforce fairness while providing acceptable throughput. It was the first partitioning scheme whose design process was guided with PBBCache. LFOC classifies
535 applications into three classes based on its cache behavior: *light sharing*, *streaming* and *cache sensitive*. LFOC reserves up to two single-way cache clusters for streaming programs. The remaining ways are distributed among cache-sensitive applications, which are then mapped to separate cache partitions whose size is determined via UCP, using as input the per-application
540 slowdown curve (i.e., IPC-based slowdown registered for different cache ways). Light sharing applications are distributed across partitions.

4.5. Notes on the simulator implementation

PBBCache was primarily designed to enable rapid prototyping and evaluation of cache-partitioning approaches. To increase programming produc-
545 tivity, it has been completely implemented in Python, and relies on libraries available for multiple operating systems. Hence it is a multi-platform tool.

To leverage parallelism in the simulator implementation we use *ipyparallel* [34]. This framework, which relies on Python's *multiprocessing* module, enables us to perform master-slave distributed-memory parallel processing.
550 The main features of *ipyparallel* are as follows:

- Master and slave (aka *engine*) processes *do not share memory*.
- The framework allows the master process to submit work (*tasks*) to be executed by one or several engines. Two complementary mechanisms are available to do so: the *Load Balanced* and the *Direct* view. With
555 the first one, the set of engines is treated as a pool of workers; the programmer does not have to explicitly determine which engine ultimately executes the task. Instead, an underlying scheduling algorithm is in place to assign tasks to engines dynamically and to quickly assign tasks to idle engines. The second mechanism, by contrast, exposes
560 individual engines to the programmer. Regardless of the mechanism used, ipyparallel allows the master to submit tasks in a synchronous

or an asynchronous way (i.e., in the latter the master does not remain blocked until the task or the set of tasks submitted completes).

- In *ipyparallel*'s programming model, engines do not communicate with each other. However, other Python modules can be used to establish explicit inter-engine communication. Because this kind of communication is required in PBBCache, we turned to the ZeroMQ messaging library [35], which is also used in ipyparallel's implementation.

The ipyparallel framework allows applications based on it to seamlessly distribute the computation across cores present in one or several machines, while maintaining a single implementation. Notably, using this approach to leverage parallelism (i.e., multiple processes that cooperate with one another) in our simulator provides much better performance and scalability on a shared-memory machine than using an ad hoc Python multithreaded application. This stems from the fact that, due to implementation issues in CPython –the reference implementation of the Python interpreter–, the truly parallel execution of multiple CPU-bound threads is not allowed within the interpreter [36]. The execution of CPU-bound threads is instead serialized, thus making multithreading an unsuitable choice for applications like our simulator. We should highlight that Jython –an alternative Python implementation written in Java– is not subject to this multithreaded-related issue. Unfortunately, its JyNi compatibility layer [37], which enables the utilization of a few Python modules written for CPython from Jython, does not currently support the *pandas* or *matplotlib* modules. Because these two widely-used modules are key building blocks of our open-source simulator [14] for seamless visualization and data manipulation, we opted to use CPython, the default Python implementation.

Finally, it is worth noting that, to solve the set of non-linear equations required for the evaluation of our bandwidth-contention model (see Sec. 4.2.1, we use the *sympy* library. This library offers a high-level and flexible mechanism to specify sets of equations in the source code; hence, making it easier to others to create and test their own models.

5. Determining the optimal solution

This section describes the features of the various B&B algorithms used by PBBCache to determine the solution of the Opt-STP and Opt-Unf optimization problems. Currently, four B&B algorithms exist: Opt-STP-S, Opt-STP-P, Opt-Unf-S and Opt-Unf-P. The name of each algorithm encodes

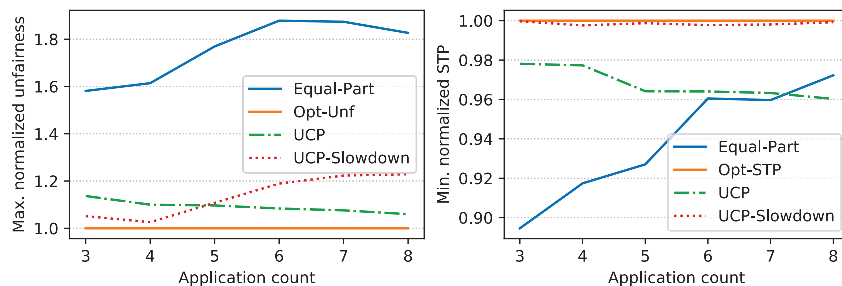


Figure 4: Comparison of various approximate algorithms and the optimal solution.

the optimization problem the algorithm solves (Opt-STP or Opt-Unf) and indicates whether the algorithm is sequential or parallel (via the -S and -P suffixes, respectively). Opt-STP-S and Opt-Unf-S follow the structure of the sequential B&B algorithm depicted in Alg. 1. They are primarily used to assess the effectiveness of the proposed bounding functions, and as a baseline to quantify the scalability of the corresponding parallel version.

All these B&B algorithms have several things in common. First, they all use best-first search. Second, the sequential and parallel algorithms for the same optimization problem share the same bounding function and the same heuristic approach to determine the initial solution. Third, despite the fact that we identify four B&B algorithms – for the sake of clarity in the explanation – two generic functions of our simulator are used to implement them. Among other things, these two functions accept as a parameter the bounding function to be applied and a flag that indicates whether it is a maximization or minimization problem. This allows us to implement the Opt-STP-P and Opt-Unf-P algorithms by invoking a single generic function; the same applies to Opt-STP-S and Opt-Unf-S. Note that this also makes it easier to extend the PBBCache with support for optimal cache partitioning under other optimization objectives (e.g., energy efficiency minimization).

In the remainder of this section we first discuss the approach to determine the initial solution (Sec. 5.1), and then proceed to present the bounding functions we use for the Opt-STP and Opt-Unf optimization problems (Sec. 5.2). Next, we describe the generic distributed-memory parallel approach used by Opt-STP-P and Opt-Unf-P (Sec. 5.3). Finally, we discuss different strategies to determine the solution of the optimal cache-clustering problem (Sec. 5.4).

5.1. Initial solution for B&B

To find a suitable strategy to determine the initial solution in the B&B algorithms, we considered 3 simple partitioning approaches: *Equal-Part*, *UCP*

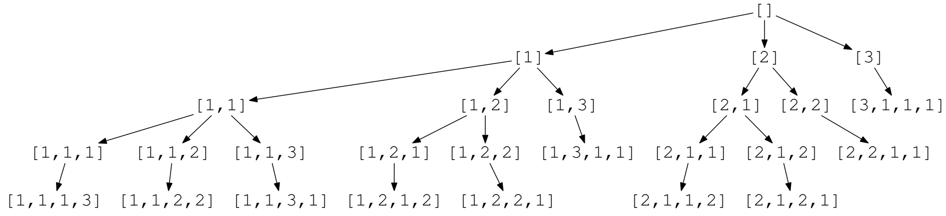


Figure 5: Search space tree for the optimal cache-partitioning problem with 4 applications and 6 ways. Eq. 9 indicates the different cases to be considered in expanding each node based on W (remaining ways) and N (remaining applications). A node has only one child (leaf) when $W{=}N$ or $N{=}1$. Otherwise it has as many as $W-N+1$ children, that come from inserting a number $\in \{1 \,..\, W-N+1\}$ at the end of the node's list.

and *UCP-Slowdown*. The first two schemes were described in Sec. 4.4; *UCP-Slowdown* is a variant of UCP [5] (approximate algorithm) that uses per-application slowdown tables (i.e. slowdown for different number of ways) instead of per-application MPKI tables. In using slowdown tables, UCP-Slowdown attempts to minimize the aggregate slowdown across applications (summation of slowdowns), by considering exclusively the performance degradation that comes from cache sharing (bandwidth contention is ignored).

Fig. 4 illustrates how the three approaches perform relative to the optimal solution for the Opt-STP and Opt-Unf optimization problems when using workloads with different application counts. Each point in the chart represents the worst value obtained for the metric in question (normalized Unfairness or STP) across 10 randomly generated workloads with the same application count (indicated on the y-axis). In light of the results, we opted to choose UCP to obtain the initial solution in the Opt-Unf-S and Opt-Unf-P B&B algorithms, as it exhibits the closest behavior to the solution of the Opt-Unf optimization problem. By contrast, for the Opt-STP-S and Opt-STP-P algorithms we employ UCP-Slowdown instead, as the STP it provides is in less than a 0.3% range of that of Opt-STP, thus clearly outperforming the other approaches. Note that both UCP and UCP-Slowdown, employ the *lookahead algorithm*, which, as reported in [5] has a worst-case time complexity of $\frac{W^2}{2}$, where W denotes the total number of cache ways.

5.2. Bounding functions

Before describing the bounding functions, we introduce the notation used to represent solutions in the optimal cache-partitioning problem. Fig. 5 shows the corresponding search-space tree for 4 applications and 6 ways. The leaf nodes of the tree represent the complete, feasible solutions available; inter-

mediate nodes represent partial solutions. For simplicity, each solution is represented as a list where each i-th item indicates the number of ways allotted to application i. So for example, the solution associated with the
655 leftmost node of the tree is `[1,1,1,3]`, namely, the first three applications in the workload get 1 way and the last one gets 3.

Henceforth, we will refer to the bounding function used by the OPT-STP-S/P algorithms as *bound_stp*, and to that of the OPT-Unf-S/P algorithms as *bound_unf*. Both bounding functions accept as a parameter the partial
660 solution (*PS*) associated with the node being explored, as well as the number of remaining ways to assign (R). The *bound_stp* function determines an upper bound of the STP value for the best solution for throughput reachable from *PS*; *bound_unf* provides a lower bound of the Unfairness metric for the best solution for fairness reachable from *PS*. Several factors make determining
665 these bounds a very complex problem. First, both metrics (see Sec. 2.1) are defined in terms of the slowdown of each application in a workload. Note that an application's slowdown depends on both the number of cache ways allotted to it, and on the degree of bandwidth contention on the system, which, in turn, varies with the distribution of the remaining cache ways among the
670 rest of applications in the workload. Second, determining the fraction of the slowdown that comes from bandwidth contention alone entails solving the set of non-linear equations of our model (Eq. 12 and 13). In our setting, this may take hundreds of milliseconds, so exploring multiple candidate solutions reachable from PS to determine a bound is largely impractical; the costly
675 bandwidth model would have to be applied multiple times (one for each candidate solution) thus incurring the associated overhead.

In defining the bounding functions, we rely on the observation that the slowdown of an application decreases as we assign more cache ways to it. Hence, the higher the number of ways available on the platform (W), the
680 higher the value of the STP metric; a higher way count also contributes to reducing unfairness in most cases. Due to the complexity of determining the bounds, coupled with the high cost associated with the bandwidth model evaluation, we opted to relax the cache partitioning problem (just for this purpose) by removing the constraint on the total number of ways that can
685 be allotted. Specifically, both bounding functions rely on determining an *ideal* solution reachable from PS that optimizes STP. The ideal solution is a feasible solution for the relaxed cache-partitioning problem, where the total number of ways assigned to the applications may be greater than W.

The ideal solution is obtained by allotting each and every application not

690 considered in the partial solution PS, the maximum amount of ways found in any feasible solution reachable from PS. Specifically, let S be the number of applications to be considered for the distribution of those remaining ways. In any possible distribution of R ways, any application gets $R-S+1$ ways at the most. Hence, the ideal solution results from completing PS by 695 assigning $R-S+1$ ways to the remaining R applications. For example, let us consider a system consisting of a 10-way cache and a workload made up of four applications. For PS=`[3,2]`, the ideal solution would be `[3,2,4,4]`.

The upper bound provided by *bound_stp* is the STP value of that ideal solution. To determine a lower bound in *bound_unf* for the Unfairness metric 700 –defined in Eq. 3– we have to follow a different approach. A trivial lower bound L for a node can be obtained as follows: $L = \frac{M}{m}$, where M and m are the maximum and minimum slowdowns, respectively, observed across applications in PS (partial solution of the node). The unfairness of any solution reachable from PS will be $\geq L$ since, in assigning ways to the rest of 705 applications, the new maximum slowdown will be $\geq M$ and the new minimum slowdown will be $\leq m$. Notably, *bound_unf* determines a less optimistic lower bound L' defined as $\frac{M'}{m}$, where M' is the maximum slowdown found in the aforementioned ideal solution (i.e., PS filled with $R-S+1$ values). Note that $L' \geq L$, and L' is still a lower bound, as the ideal solution guarantees the 710 lowest possible slowdown for the remaining applications, hence minimizing the ratio, as we keep the same denominator m.

Despite the simplicity of the bounding approach, our experimental results in Sec. 6.3 reveal that the *bound_stp* and *bound_unf* functions lead to very effective pruning. Moreover, because the bandwidth model has to be evaluated 715 just once, the approach is affordable in terms of computational cost.

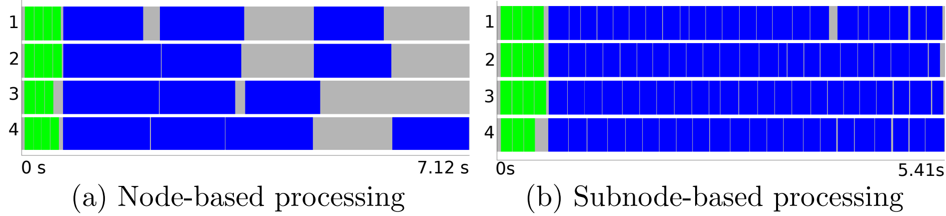
5.3. Parallel distributed-memory B&B algorithms

The Opt-STP-P and Opt-Unf-P algorithms follow the same parallel strategy. For the sake of simplicity in the explanation, we will describe the strategy assuming a minimization problem, as it is done in [24].

720 We begin by discussing the main challenges that arise in attempting to solve the Opt-Unf and Opt-STP problems via parallel B&B. First, as shown in Fig. 5, the search space tree is largely unbalanced. A possible approach to parallelizing the search consists in breaking down the full tree into subtrees – preferably with a similar node count, and processing these subtrees in par- 725 allel. However, this approach does not necessarily provide good scalability

due to the unpredictable effect of pruning; the number of nodes of a particular subtree that ultimately have to be processed depend on the pruning effectiveness in that area of the search space. Therefore, considering a entire subtree as the work unit for parallel processing does not constitute a good approach, as it leads to load imbalance. Secondly, calculating the cost of a solution (leaf node of the tree) or determining the lower bound of any node entails solving the set of non-linear equations of the bandwidth model, which, as stated earlier, may have a substantial overhead (in the order of hundreds of ms.). Note that when processing a node of the tree, these tasks usually have to be performed several times. Because this kind of processing has enough computational complexity, treating one individual node (rather than a subtree) as the work unit for parallel computation constitutes a promising approach. Our strategy is based on this idea.

Our distributed-memory parallel approach follows the master-slave pattern. In our B&B approach the work unit to be processed by slave processes is the *subnode*; we use this term to denote the processing that has to be done for a subset of children of a certain node in the tree. Recall that in processing a node, the B&B method has to determine the lower bound for all of its children; children nodes with a lower bound greater than the upper bound are pruned, and the remaining nodes are either considered when updating the upper bound (leaf nodes) or left for further processing. We observed that using the node as the work unit leads to load imbalance, as the higher the number of children – which largely varies across nodes of the tree – the higher the computational cost associated with processing the node. To overcome this problem, we divide the node-level processing into groups of children, each one with a children count not greater than `max_children` – a configurable parameter of our algorithm. Specifically, a node consisting of C children is divided into $\lceil \frac{C}{\texttt{max_children}} \rceil$ subnodes. Breaking down the node-level processing in this manner allows us to create smaller tasks with similar granularity, which enable a more even distribution of the work especially when pruning is working very effectively (just a few nodes to process). To illustrate this fact, Figs. 6a and 6b show the granularity of tasks that come from using the node and the subnode as the work unit. Clearly, using subnodes leads to a higher number of smaller, more uniform tasks. This contributes to reducing load imbalance and improves performance. The default value of the `max_children` parameter is 3, which makes it possible to obtain fine-grained tasks but with enough computational complexity so that it is worth it to send them to slaves for processing even on a remote machine (like the setting



(a) Node-based processing (b) Subnode-based processing

Figure 6: Traces for Opt-STP-P obtained with Paraver [38] (6 applications and 4 slave processes). Tasks in blue denote node (a) and subnode (b) processing; idle periods appear in gray; light-green tasks represent the parallel initialization of the subnode queue.

explored in Sec. 6).

Algorithms 2 and 3 outline the behavior in a minimization B&B of the master process and the subnode processing by a slave. The master submits subnodes to the slave pool; upon completion of a subnode processing request the slave process returns a list of promising child subnodes back to the master – to be processed later. The distributed algorithm terminates when there are no subnodes left to process. Because master and slave processes do not share memory, each process keeps a local copy of the upper bound; when any process finds a better solution, the new identified upper bound is notified to the rest of the processes by using the publish-subscribe communication pattern of the ZeroMQ messaging library [35]. Specifically, each process acts as an independent publisher and, in turn, is subscribed to the notifications issued by the remaining processes. Upon receiving a notification, if the remote upper bound is better than the local one, the process updates the local upper bound, which will be used for pruning from then on. Note that using ZeroMQ for this task provides a simple and efficient implementation, and does not create additional library dependencies for our simulator, as ipyparallel's implementation already relies on ZeroMQ.

The master process maintains a priority queue of subnodes yet to be submitted (line 8), and a list with pending subnodes currently being processed by slaves (line 13). Subnodes in both data structures are sorted in ascending order by its lower bound, so as to perform pruning operations more efficiently. To follow a best-first approach, the most promising subnode, located at the front of the queue is submitted first. When the upper bound is updated in the master process unpromising subnodes in the queue are simply pruned by removing them from the queue (line 17). The master may also prune unpromising subnodes being currently processed by slaves; the corresponding task in the slave process is immediately canceled remotely and the associated subnode is removed from the pending list (line 20).

Algorithm 2: Simulator's master B&B code.

```
Input: A is the workload considered for cache partitioning, W number of cache ways available
       on the platform, slave_count is the number processes in slave_pool
1  incumbent ← initial solution for A provided by heuristic algorithm (see Sec. 5.1)
2  upper_bound ← incumbent.Cost();
3  U ← slave_count * initial_load ; queue_limit ← slave_count * 2 ; pending ← [ ]; prio_q ← [ ]
4  Send A and other global data to slave_pool
   // Initialization of the subnode queue (prio_q)
5  NS ← unroll a number of nodes no smaller than U from A's tree via breadth-first traversal
6  Calculate lower bound in parallel for each node N_i ∈ NS using slave_pool
7  Remove nodes from NS whose lower bound ≥ upper_bound (those will not be processed)
8  foreach node n_i in NS do prio_q.concat(break_into_subnodes(n_i, W, max_children)) end
9  while !prio_q.isEmpty() || !pending.isEmpty() do
       // Submit subnodes to the slave pool asynchronously
10     while !prio_q.isEmpty() && len(pending) ≤ queue_limit do
11         subnode ← pop highest prio subnode from prio_q
12         ptask ← slave_pool.async_submit(subnode, subnode.lower_bound)
13         pending.append(ptask, subnode.lower_bound)
14     end
15     completed_tasks ← Remain blocked until at least one task in pending list completes
16     (local_ub, local_inc) ← (upper_bound, incumbent)
       // Retrieve remote upper bounds (ZeroMQ), update and prune if necessary
17     (incumbent, upper_bound) ← process_remote_info(prio_q, pending, local_ub, local_inc)
       // Process completed tasks
18     foreach t_i in completed_tasks do
19         foreach subnode s_j in t_i.promising_child_subnodes() do append s_j to prio_q if
             s_j.lower_bound < upper_bound end
20         Remove t_i from pending
21     end
22 end
```

As shown in Alg. 2, the master process continuously submits subnodes to the slave pool asynchronously. The load balancing algorithm of the ipy-
795 parallel framework [34] automatically maps work to specific slaves so as to balance the load in the slave pool. The submission of subnodes is temporarily stalled by the master when the total number of subnodes being processed by slaves exceeds twice the number of slave processes. We found that increasing the number of pending subnodes beyond that point leads to submitting a
800 higher number of potentially unpromising subnodes, that would have been otherwise pruned locally by the master, rather than canceled. This degrades performance as it may keep slave processes busy for a longer period of time doing useless work. Conversely, considering fewer pending subnodes causes slaves to go idle more frequently, as they wait more often for the master
805 to submit new work, thus negatively impacting scalability. Therefore, our choice of the maximum number of pending tasks provides a good trade-off between pruning effectiveness and scalability.

Finally, we zoom in on the initialization of the subnode queue (lines 3-8 –

Algorithm 3: Simulator's slave B&B code for subnode processing.

Input: A: workload considered for cache partitioning; W number of cache ways available on the platform; *subnode* to be processed, LB: lower bound of *subnode*. (Note that *upper_bound* and *incumbent* are global variables, but local to each slave process)

```
1  promising ← [ ] // Initialize list of promising subnodes to be returned to master
2  if subnode.isSolution() || subnode.ReachableSolutions() == 1 then
3      (solution, cost) ← subnode.getSolution()
4      if cost < upper_bound then
5          (incumbent, upper_bound) ← (solution, cost)
6          zeromq_publish(incumbent, upper_bound)
7      end
8  else
9      foreach child in subnode.getChildren() do
10         (local_ub, local_inc) ← (upper_bound, incumbent)
           // Retrieve remote upper bounds (ZeroMQ), update and prune if necessary
11         (incumbent, upper_bound) ← process_remote_info(promising, [ ], local_ub, local_inc)
12         if LB > upper_bound then return [ ]
13         if child.isSolution() || child.ReachableSolutions() == 1 then
14             (solution, cost) ← child.getSolution()
15             if cost < upper_bound then
16                 (incumbent, upper_bound) ← (solution, cost)
17                 zeromq_publish(incumbent, upper_bound)
18             end
19         else
20             lower_bound ← bounding_function (child);
21             child.setLowerBound(lower_bound);
22             if lower_bound < upper_bound then
23                 (incumbent, upper_bound) ← (solution, cost)
24                 promising_q.concat(break_into_subnodes(child, W, max_children))
25             end
26         end
27     end
28 end
29 return promising
```

Alg. 2). A simple way to initialize it would be to insert the root node of the tree only, as done in the sequential algorithm depicted in Alg. 1. However, using this approach here degrades scalability substantially as it does not keep all slave processes busy from the beginning of the execution. Specifically, it takes some time to expand enough subnodes to make this happen. Our algorithm instead unrolls a certain number of tree nodes via breadth-first traversal (line 5). In doing so, the master builds a list of nodes that belong to a certain level of the tree l or to two consecutive levels (l and $l+1$); nodes in the upper levels of the tree ($< l$) are automatically discarded by the B&B algorithm so as to remove the substantial overhead associated with the evaluation of the bounding function in the master process. Nodes on the list are submitted to slave processes so as to compute their lower bound in parallel; nodes whose lower bound is higher than the upper bound are pruned.

Finally, promising nodes are broken down into subnodes (line 8), which are used to populate the queue. Note that the number of nodes unrolled is no smaller than `initial_load` $*$ `slave_count`, where `slave_count` denotes the
825 number of slave processes, and `initial_load` is a configurable parameter of the algorithm, whose default value is 2. That value typically ensures that the queue is initialized with enough subnodes to keep slaves busy.

5.4. Determining the optimal cache-clustering solution

To determine the optimal clustering for a workload A, we must consider
830 all *cluster sets* of A. For each cluster set, the optimal distribution of the cache space among clusters must be determined. The optimal solution is the best one observed across cluster sets.

In this problem, each possible cluster set can be explored in parallel. In turn, we may also exploit parallelism to determine the optimal cache parti-
835 tioning for a given cluster set. Exploiting these two complementary levels of parallelism, however, is not possible with ipyparallel, as nested parallelism is not currently supported. PBBCache implements a master-slave algorithm that evaluates multiple cluster sets in parallel. The master generates every possible cluster set for the workload and submits them to the slave pool for
840 processing by leveraging a mechanism to restrict the number of pending tasks similar to the one described in Sec. 5.3. Slaves employ the Opt-Unf-S or Opt-STP-S algorithms to sequentially determine the optimal cache partitioning for a given cluster set under the fairness and STP optimization objectives, respectively. Finally, it is worth highlighting that the exploitation of the
845 outermost level of parallelism (what we do here) is generally more effective than considering the inner one, because in many of the cluster sets that must be explored, the corresponding search space tree for the optimal cache-partitioning problem is very small. Processing these small trees in parallel does not bring substantial performance gains relative to doing it sequentially.

850 6. Experiments

6.1. Experimental Setup

To carry out the evaluation of various aspects of the PBBCache simulator we conducted experiments on different platforms with GNU/Linux, and using the 4.9.116 version of the Linux kernel. Table 1 summarizes the features of the
855 four hardware platforms we used, which include different models of the Intel Xeon processor family. Essentially, Platforms A and B, which are the only ones equipped with Intel CAT, were used to extract performance information

Table 1: Features of the various platforms used

Platform Name	**A**	**B**	**C**	**D**
Proc. Model	Xeon E5-2620 v4	Xeon Gold 6138	2 x Xeon E5-2695 v3	2 x Xeon E5-2650
Proc. Frequency	2.1GHz	2.0 GHz	2.3 GHz	2.0GHz
Core count	8	20	28	16
LLC (L3) cache	20MB/20-way	35MB/11-way	35MB/20-way	20MB/20-way
Main Memory	32GB@2133 MHz	96GB@2666 MHz	64GB@1600 MHz	64GB@1600 MHz
CAT enabled	Yes	Yes	No	No

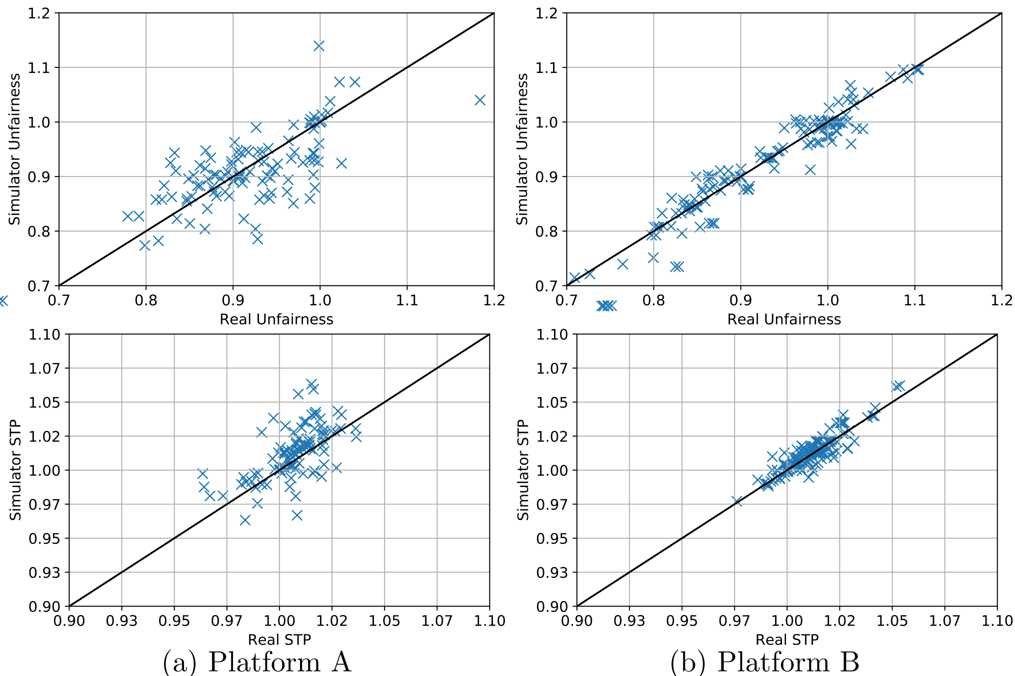
from various benchmarks to be used as input for the simulator. We also used these two platforms to carry out the simulator validation (Sec.6.2). Platforms A and B integrate processors with different microarchitectures (Broadwell and Skylake, respectively), different organizations of the cache hierarchy (e.g. Platform A uses an inclusive LLC whereas B does not) and different cache partitions granularities (i.e. the smallest partition we can create in B – 2.5 megabytes – is 2.5 times bigger than on A).

To assess the effectiveness of pruning in our B&B algorithms (Sec. 6.3), and measure the scalability of the parallel B&B strategy on one node (Sec. 6.4), we turned to Platform C, which features more cores than Platforms A and B. Finally, we experimented with a cluster consisting of four 16-core nodes (each one with the features shown in Table 1 for Platform D) to demonstrate that the simulator can be effectively executed on a cluster as well.

6.2. Validation of the simulator

For the validation experiments we used 25 randomly generated workloads consisting of 6 and 8 programs each. In building the workloads, which exhibit a different degree of shared-resource contention, we used 25 different benchmarks from SPEC CPU. For each workload we collected the STP and Unfairness values provided by PBBCache for various partitioning algorithms: UCP [5], Yu-Petrov [10], Equal-Part, KPart [9] and LFOC [8]. We also gathered the corresponding values for the optimal solutions provided by PBBCache for the Opt-Unf and Opt-STP problems. To validate these results we compared them with those observed when applying the same partitioning statically on the real machine where the corresponding performance information for the simulation was obtained (Platforms A and B).

For the experiments on the real machine, we rely on the PMCTrack tool [39], which makes it possible to establish per-process cache partitions from user-space on systems equipped with Intel CAT. Essentially, prior to the execution of the workload under a certain partitioning approach we run the simulator to retrieve the associated cache partitions, and map each applica-



(a) Platform A (b) Platform B

Figure 7: Real vs. simulator-provided values for the STP and Unfairness metrics on platforms A (left) and B (right), normalized to the results of the Equal-Part scheme.

tion in the workload to its corresponding partition. Note that, for simplicity, the partitions remain the same (static) throughout the workload's execution. We ensure that all applications in the mix are started simultaneously and when one of them terminates it is restarted repeatedly until the longest application in the set completes three times. We then measure unfairness and STP, by using the geometric mean of the completion times for each program.

Figs. 7a and 7b show the comparison of the real vs. simulator-provided values for the STP and Unfairness metrics on platforms A and B, respectively. Both metrics have been normalized to the results of the Equal-Part scheme. We observe that the Unfairness and STP values provided by the simulator closely track those of the actual platforms. Specifically, the average error rate observed on Platform A (Broadwell architecture) is 3% and 1% for Unfairness and STP respectively, and on the Platform B (Skylake architecture) is 2% and 0.4%. We found that the main reason behind the less accurate simulations on Platform A has to do with inaccuracies in the bandwidth model for some bandwidth-intensive applications. On this platform, the theoretical maximum memory bandwidth is 68.3GB/s, whereas on Platform B this bandwidth is 128GB/s. As a result, bandwidth contention is

substantially higher on the first platform. In particular, these model inaccuracies are due to the fact that PBBCache was fed with the average bandwidth registered across the execution to predict the slowdown due to contention. In some cases, this average does not represent the benchmark behavior in certain program phases where we find spikes in bandwidth consumption that are substantially higher than the average. Consequently, the actual slowdown is underpredicted due to serious bandwidth contention in these cases, leading to lower Unfairness and STP values.

A potential way to address this issue, which did not prevent us from gathering good predictions on Platform B, would be to make PBBCache aware of the different program phases of bandwidth-intensive applications. Because phase transitions do not occur at the same time in multiple programs, this would require to break down the whole workload execution into stages of stable behavior across applications, and apply the particular partitioning scheme on each stage. Unfortunately, this approach would make it difficult to obtain a solution in a reasonable amount of time, which stands in contrast with the main goal of the simulator: a tool for rapid prototyping and evaluation. Note that to determine an exact solution the B&B algorithm for the Opt-STP (or Opt-Unf) non-linear optimization problem would have to be invoked for each execution stage one after another, as the progress each application makes (which must be determined to detect the next phase change) depends in turn of the partitioning applied in the previous stage. In our earlier work [8] we demonstrated that PBBCache proves very useful in aiding in the design of novel partitioning algorithms; the aforementioned innacuracies associated with bandwidth-intensive applications did not prevent us from using PBBCache to guide the design process of the LFOC approach [8], which outperforms previous fairness-aware partitioning proposals.

Fig. 8 shows the accuracy of the simulator predictions on Platforms A and B for some representative workloads, which summarize the trends observed in all our experiments. The results are reported separately for each individual partitioning scheme. Clearly, PBBCache is capable of capturing the relative benefit in STP and Unfairness that a partitioning scheme obtains over the others, and enables us to identify the best performing schemes in each scenario. Notably, dividing the LLC into same-sized partitions (Equal-Part) does not constitute a good approach regarding fairness or throughput, as, in doing so, we do not cater to the degree of cache sensitivity of each application. Yu-Petrov's algorithm provides better results in a few cases but is also subject to high fairness/throughput degradation. Moreover, we

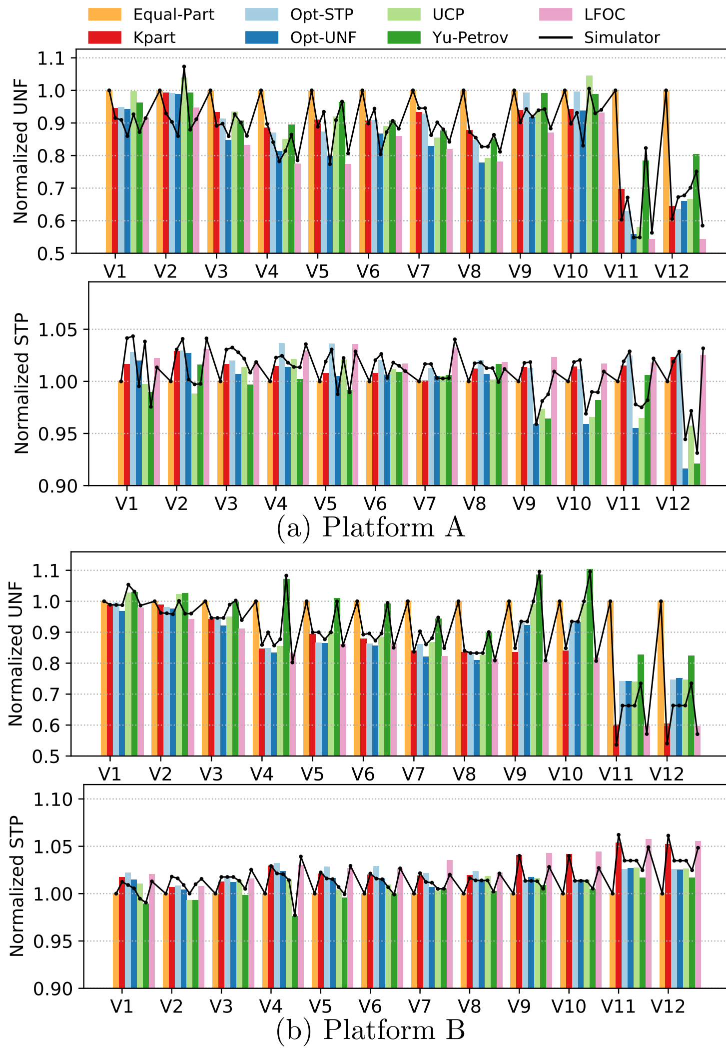


Figure 8: Real vs. simulator-provided values for the STP and Unfairness on Platforms A and B, normalized to the results of Equal-Part.

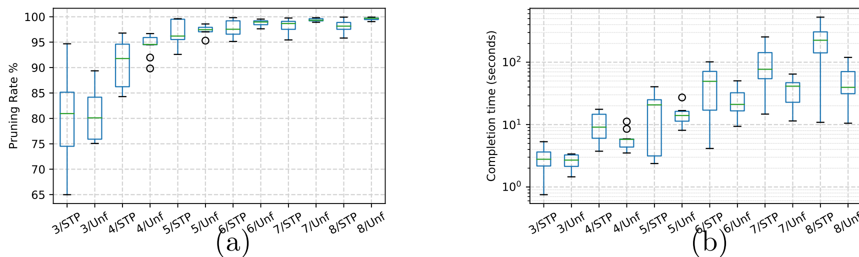


Figure 9: Pruning rate (a) and completion time (b) for sequential B&B algorithms under different sets of workloads. Labels X axis, with format *n/target* indicate the number of applications in the workload (n) in the corresponding set, and the optimization metric.

observe that UCP and KPart are more effective in general, but are still far from the optimal cache-partitioning solutions Opt-Unf and Opt-STP in most cases. As pointed out in previous work [4, 8], cache-clustering policies can outperform strict cache-partitioning policies. This is the case for workloads V9-V12 on Platform B, where the KPart cache-clustering policy provides better throughput and fairness than Opt-STP and Opt-Unf. Finally, LFOC is able to obtain lower unfairness values than the other approaches across the board, and also reaps higher or comparable throughput (STP) to many other policies in most cases. To sum up, cache-clustering policies are in general superior to strict cache-partitioning approaches on recent Intel server platforms, where the smallest cache partitions that can be created have a relatively coarse granularity (e.g., 1 megabyte on Platform A).

6.3. Effectiveness of the bounding functions Opt-STP and Opt-Unf

One the key aspects of the proposed B&B algorithms is the effectiveness of the *bound_stp()* and *bound_unf()* bounding functions, described in Sec. 5.2, and used for the Opt-STP and Opt-Unf problems, respectively. To compare the efficacy of both bounding functions and analyze their impact in performance we randomly built 66 workloads consisting of a number of applications that range between 3 and 8 (the core count of Platform A). Eleven workloads were considered for each application count. For these experiments we fed PBBCache with data gathered on Platform A where the number of possible solutions is substantially higher than on Platform B (see Sec. 2.2).

Fig. 9 shows the pruning rate and the completion time for the different workload sets (one for each application count) and optimization metrics (i.e. STP and Unfairness) obtained by the Opt-STP-S and Opt-Unf-S sequential B&B algorithms. The pruning rate is defined as the percentage over the total

970 number of nodes in the search space tree that were discarded by pruning. The results reveal that the pruning rate largely depends on the nature of the workload. For example, for 3-application workloads under Opt-STP-S, the pruning rate ranges between 65% and 94.7%. We also observe that, as we increase the number of applications in the workload, the variability decreases, 975 and the average pruning rate improves substantially; it is greater than 96.7% when the application count is >5. This indicates that the effectiveness of the bounding functions increases with the problem size, which is a good property of our proposed B&B algorithms.

As expected, the pruning rate has an enormous impact on the completion 980 time. In particular, the slight superiority of *bound_unf()* over *bound_stp()* for 6-8 applications (see Fig. 9a) leads to substantially smaller completion times due to pruning (Fig. 9b). Notably, because the number of nodes in the search space tree – shown in Table 2 – grows exponentially with the application count, a small increase in the pruning rate may have a tremendous impact 985 on the completion time. Specifically, an increase of 0.5% of the pruning rate for an 8-application workload can accelerate the execution by a factor of 2x. All in all, we observe that for the workloads explored the Opt-Unf-S algorithm provides the optimal solution in less than 2 minutes, and the Opt-STP-S algorithm in less than 9 minutes. Given the good scalability of the 990 parallel B&B approach (as we discuss next) these completion times can be reduced substantially with the parallel B&B algorithms.

Application count	**3**	**4**	**5**	**6**	**7**	**8**
Number of nodes	357	2056	8295	24955	58071	106963

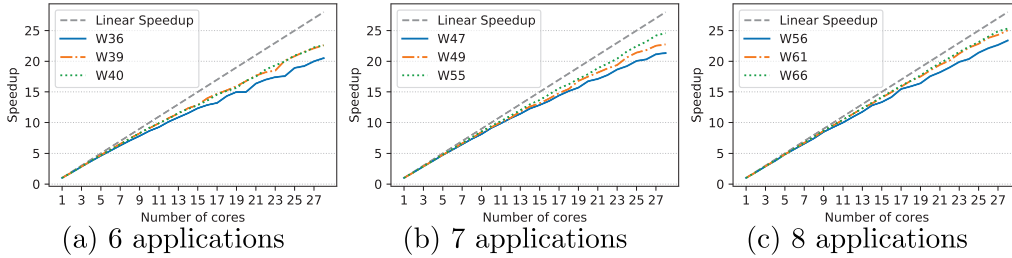
Table 2: Number of nodes of the search space tree for different workloads on Platform A.

6.4. Scalability of the distributed-memory parallel B&B strategy

6.4.1. Single-node results

To assess the scalability of proposed parallel B&B strategy on a single 995 multicore server we used Platform C (28 cores, divided into two sockets, as shown in Table 1). Specifically, we focused on the study of a subset of the workloads explored in Sec. 6.3 where the Opt-STP-S (sequential) algorithm takes more than one minute to find the optimal solution.

Fig. 10 shows the speedup achieved with the Opt-STP-P algorithm for 1000 the different application mixes explored (consisting of 6, 7 and 8 applications, respectively) as we vary the number of cores from 1 to 28. As stated in Sec. 5.3, the `max_children` and `initial_load` parameters of the algorithm



(a) 6 applications (b) 7 applications (c) 8 applications

Figure 10: Scalability for different workload sets consisting of 6, 7 and 8 applications

were set to 3 and 2, respectively. Note that the speedup is reported relative to the completion time of the sequential B&B algorithm. The linear speedup was also included in the charts for comparison purposes.

The results reveal that the maximum speedup registered is 25.2x (W55 with 28 cores), which corresponds to a parallel efficiency of 0.9. Overall, we also observe that the speedup of our parallel approach gets closer to the linear speedup as the workload size increases. This is a positive trend, since the parallel strategy becomes more effective in utilizing multiple cores as the sequential B&B algorithm begins to exhibit substantially longer completion times (up to 9 minutes as shown in Fig. 9b for the STP metric). Hence, by leveraging parallelism on this platform our proposed simulator is able to determine the optimal solution for any of these workloads in less than 34s. Finally, we should highlight that the reason of the lower scalability observed for 6-application workloads (Fig. 10a) has to do with the fact that at the end of the execution of the parallel algorithm the number of remaining subnodes to process is smaller than the number of cores, leaving a few cores idle for a short time period of time. As the problem size increases (higher application count) the fraction over the total execution affected by this imbalance scenario is smaller, which provides better scalability. We next discuss further this aspect as it also becomes apparent in our multi-node experiments.

6.4.2. Multi-node results

For the evaluation of the parallel B&B strategy using multiple computing nodes, we used a cluster consisting of four identical 16-core server systems that follow the specifications of Platform D, as described in Table 1. Fig. 11a reports the speedup observed for different workloads as we increase the number of cluster nodes from 1 to 4 (i.e. from 16 to 64 cores).

The results reveal that the parallel B&B strategy is capable of obtaining substantial performance gains relative to the sequential approach by effectively utilizing multiple cores on different nodes. We also observe that the

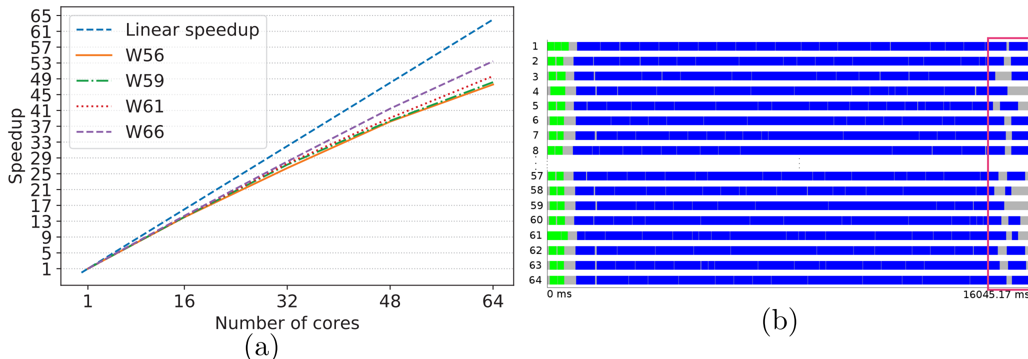


Figure 11: (a) Speedup for different workload sets using from one to four nodes (16 cores each) on Platform D. (b) Excerpt of the execution trace for W66 with 64 cores. Note that idle periods are denoted in gray as in traces shown in Sec. 5.3

speedup achieved for up to two nodes (32 cores) is very close to the linear speedup, but it drops slightly when using 3 or more nodes. Specifically, for three nodes (48 cores) the parallel efficiency for the various workloads ranges between 0.8 and 0.87. We found that this trend is caused by the issue described in Sec. 6.4.1; as the number of cores increase, some of them remain idle for a short period of time at the end of the execution due to the shortage of subnodes to process. Fig. 11b illustrates this fact by means of a sample execution trace of the algorithm with 64 cores. We observed that in increasing the problem size, the speedup gets closer to its linear counterpart. A potential approach to obtain higher scalability in maintaining the problem size constant is to start using finer-grained subnodes as the algorithm begins to reach the end of the execution. That would entail (1) maintaining a counter in the master process that keeps track of the number of feasible solutions remaining to explore – by leveraging Eq. 9 – and (2) devising a policy to adjust the value of the `max_children` parameter dynamically as we get closer to the end of the execution. We plan to implement that promising optimization as part of a future release of our open-source simulator [14].

7. Conclusions and Future Work

In this article we have presented PBBCache, an open-source [14] parallel simulator written in Python enabling rapid prototyping and evaluation of cache partitioning and clustering policies. PBBCache allows researchers to quickly compare novel approaches with the optimal solution or with existing cache-partitioning schemes, making it possible to determine if a new approach

1055 is promising even before starting its implementation in the system software and evaluation on real hardware, which can be a time-consuming task [8].

A key aspect of our simulator is the mechanism it employs to determine the relative performance degradation (i.e. slowdown) that an application suffers when running simultaneously with others on a multicore system. This 1060 mechanism factors in the slowdown that comes from contention on the two main critical shared resources: the last-level cache and memory bandwidth. To account for memory bandwidth contention we extended the model proposed in [11] with awareness on how sensitive application performance is to a reduction of the available bandwidth. Evaluating this model entails 1065 solving a set of non-linear equations, which is computationally expensive, and makes determining the optimal cache partitioning that maximizes system throughput or fairness a mixed-integer non-linear problem. To efficiently solve these optimization problems by exploiting parallelism, PBBCache leverages a distributed-memory branch-and-bound (B&B) strategy specifically 1070 tailored for optimal cache partitioning. The mechanism used by this B&B approach to break down the work to be done in parallel into tasks with a similar computational complexity, coupled with the effectiveness of the bounding functions specifically designed for the optimization problems considered, gives rise to a scalable strategy that effectively utilizes multiple cores 1075 on one or several computing nodes, as we demonstrate in this work.

For the validation of PBBCache's simulation model we conducted experiments on real platforms that support Intel CAT and Memory Bandwidth Monitoring. Our analysis reveals that the simulator succeeds in identifying what partitioning approach is the most effective for particular workloads. 1080 Notably, in a recent work [8] we demonstrated that PBBCache makes an effective tool to aid in the design of novel cache partitioning policies; the effectiveness of the LFOC fairness-oriented approach – proposed in [8] and evaluated in our validation experiments – constitutes a clear example of the potential of the simulator. An interesting avenue of future work is augment-1085 ing PBBCache with support to approximate the energy efficiency delivered by a certain partitioning scheme, as that would make it possible to aid in the design of energy-aware approaches. Similarly, we plan on devising a model to factor in the effect of applying per-application memory-bandwidth consumption caps via hardware mechanisms such as the recent Intel Memory 1090 Bandwidth Allocation (MBA) technology, so as to enable rapid evaluation of policies that simultaneously exploit the Intel CAT and MBA features. We will also explore alternative ways to determine the optimal solution more

efficiently via B&B by employing hybrid depth-first and best-first schemes.

Acknowledgements

1095 We thank the anonymous reviewers for their invaluable feedback. This work has been supported by the EU (FEDER), the Spanish MINECO and CM, under grants TIN 2015-65277-R, RTI2018-093684-B-I00 and S2018/TCS-4423. Adrian Garcia-Garcia is supported by a UCM fellowship grant.

Appendix A. Formalization of Opt-STP and Opt-Unf as MINLPs

1100 Here we provide a detailed formalization of the Opt-STP and Opt-Unf problems for a workload A consisting of N single-threaded applications $\{a_1, a_2, \cdots, a_N\}$ that run on a system featuring a W-way last-level cache with $W \geq N$.

We first present the parameters and decision variables which are common to both optimization problems. The **parameters** represent performance 1105 metrics of each application in the workload. These values are gathered offline as an application runs alone on the system under different way assignments:

$$IPC_{a \in A, k \in K} \quad \textit{IPC alone} \tag{A.1}$$

$$B_{a \in A, k \in K} \quad \textit{Bandwidth alone} \tag{A.2}$$

$$TS_{a \in A, k \in K} \quad \textit{Total stalls} \tag{A.3}$$

$$MS_{a \in A, k \in K} \quad \textit{Memory stalls} \tag{A.4}$$

Other parameters are calculated internally based on the others:

$$pIPC_a = IPC_{a,W}, \forall a \in A \quad \textit{IPC with all available ways} \tag{A.5}$$

$$pS^c_{a,k} = \frac{pIPC_a}{IPC_{a,k}}, \forall a \in A, \forall k \in K \quad \textit{Relative performance degradation} \tag{A.6}$$

1110 The **decision variables** are the following:

$$w_{a \in A, k \in K} \in \{0, 1\} \quad \textit{Way assignment} \tag{A.7}$$

$$\overline{vB}_{a \in A} \in \mathbb{R} \quad \textit{Effective Bandwidth alone} \tag{A.8}$$

$$\overline{vB'}_{a \in A} \in \mathbb{R} \quad \textit{Bandwidth shared} \tag{A.9}$$

$$\overline{vT} \in \mathbb{R} \quad \textit{Total bandwidth shared} \tag{A.10}$$

$$vSC_{a \in A} \in \mathbb{R} \quad \textit{Effective slowdown due to cache-sharing} \tag{A.11}$$

$$vSB_{a \in A} \in \mathbb{R} \quad \textit{Bandwidth Slowdown} \tag{A.12}$$

$$vTS_{a \in A} \in \mathbb{R} \quad \textit{Effective total stalls} \tag{A.13}$$

$$vMS_{a \in A} \in \mathbb{R} \quad \textit{Effective memory stalls} \tag{A.14}$$

$$vS_{a \in A} \in \mathbb{R} \quad \textit{Effective Slowdown} \tag{A.15}$$

The **Opt-STP** problem can be formulated as follows:

$$\text{Maximize}: \ \ \sum_{a \in A} \frac{1}{vS_a} \tag{A.16}$$

subject to these linear constraints:

$$\overline{vT} = \sum_{a \in A} \overline{vB\text{'}}_a \tag{A.17}$$

$$\sum_{k \in K} w_{a,k} = 1, \forall a \in A \tag{A.18}$$

$$\sum_{a \in A} \sum_{k \in K} k \cdot w_{a,k} = W \tag{A.19}$$

$$1 \leq \sum_{k \in K} k \cdot w_{a,k} \leq W - N + 1, \forall a \in A \tag{A.20}$$

$$\overline{vB}_a = \sum_{k \in K} B_{a,k} \cdot w_{a,k}, \forall a \in A \tag{A.21}$$

$$vSC_a = \sum_{k \in K} pS^c_{a,k} \cdot w_{a,k}, \forall a \in A \tag{A.22}$$

$$vTS_a = \sum_{k \in K} TS_{a,k} \cdot w_{a,k}, \forall a \in A \tag{A.23}$$

$$vMS_a = \sum_{k \in K} MS_{a,k} \cdot w_{a,k}, \forall a \in A \tag{A.24}$$

And also **subject to** the following set of non-linear constraints for the evaluation of the bandwidth model. Specifically, $\forall a \in A$, we have that:

$$(\overline{vB\text{'}}_a)^2 \cdot \left(1 - \frac{1}{\overline{\overline{vB}}_a}\right) + \overline{vB\text{'}}_a \cdot \left(1 - \frac{1}{\overline{\overline{vT}}}\right) \cdot \left(1 - \frac{1}{\overline{\overline{vB}}_a}\right) + 1 - \frac{1}{\overline{\overline{vT}}} = 0 \tag{A.25}$$

$$vSB_a = \frac{vTS_a + vMS_a \cdot \left(\frac{\overline{vB}_a}{\overline{vB\text{'}}_a} - 1\right)}{vTS_a} \tag{A.26}$$

$$vS_a = vSC_a \cdot vSB_a \tag{A.27}$$

The **Opt-Unf** problem can be formulated as a MINLP, as follows:

$$\text{Minimize}: \ \ \frac{S_{\max}}{S_{\min}}, S_{\max} \geq vS_a, \forall a \in A, S_{\min} \leq vS_a, \forall a \in A \tag{A.28}$$

subject to the constraints specified by Equations A.17 to A.27.

References

[1] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, M. Prieto-Matias, Survey of scheduling techniques for addressing shared resources in multicore processors, ACM Comput. Surv. 45 (1) (2012) 4:1–4:28.

[2] E. Ebrahimi, C. J. Lee, O. Mutlu, Y. Patt, Fairness via source throttling:

a configurable and high-performance fairness substrate for multi-core memory systems, in: Proc. of ASPLOS'10, 2010, pp. 335–346.

[3] A. Garcia-Garcia, J. C. Saez, M. Prieto-Matias, Contention-aware fair scheduling for asymmetric single-ISA multicore systems, IEEE Transactions on Computers 67 (12) (2018) 1703–1719.

[4] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, M. E. Gómez, Application clustering policies to address system fairness with Intels Cache Allocation Technology, in: Proc. of PACT'17, 2017, pp. 194–205.

[5] M. K. Qureshi, Y. N. Patt, Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches, in: Proc. of MICRO'06, 2006, pp. 423–432.

[6] S. Mittal, A survey of techniques for cache partitioning in multicore processors, ACM Comput. Surv. 50 (2) (2017) 27:1–27:39.

[7] K. Nguyen, Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family, `https://software.intel.com/en-us/articles/introduction-to-cache-allocation-te` (2016).

[8] A. Garcia-Garcia, J. C. Saez, F. Castro, M. Prieto-Matias, LFOC: A lightweight fairness-oriented cache clustering policy for commodity multicores, in: Proc. of ICPP'19, 2019.

[9] N. El-Sayed, A. Mukkara, P. Tsai, H. Kasture, X. Ma, D. Sanchez, KPart: A hybrid cache partitioning-sharing technique for commodity multicores, in: Proc. of HPCA'18, 2018.

[10] C. Yu, P. Petrov, Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms, in: Proc. of DAC'10, 2010.

[11] T. Y. Morad, N. Shalev, I. Keidar, A. Kolodny, U. C. Weiser, EFS: Energy-Friendly Scheduler for memory bandwidth constrained systems, Journal of Parallel and Distributed Computing 95 (2016) 3 – 14.

[12] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, O. Mutlu, The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory, in: Proc. of MICRO'15, 2015, pp. 62–75.

[13] R. Love, Linux Kernel Development, 3rd Ed., Addison-Wesley, 2010.

[14] A. Garcia-Garcia, J. C. Saez, J. Casas, Source code repository. PBBCache: A parallel branch-and-bound based cache-partitioning simulator, https://github.com/pbbcache/cachesim (2019).

[15] S. Cass, Interactive: The Top Programming Languages 2019, https://spectrum.ieee.org/computing/software/the-top-programming-languages-20 (2019).

[16] SCIP: solving constraint integer programs, https://scip.zib.de/, accessed: 2019-07-18.

[17] BONMIN: Basic open-source nonlinear mixed integer programming, https://www.coin-or.org/Bonmin/, accessed: 2019-07-17.

[18] BARON: A general purpose global optimization software package, http://archimedes.cheme.cmu.edu/?q=baron, accessed: 2019-07-15.

[19] The NEOS server, https://neos-server.org/neos/, accessed: 2019-07-18.

[20] D. Xu, C. Wu, P.-C. Yew, J. Li, Z. Wang, Providing fairness on shared-memory multiprocessors via process scheduling, in: Proc. of SIGMETRICS'12, 2012, pp. 295–306.

[21] S. Eyerman, L. Eeckhout, System-level performance metrics for multiprogram workloads, IEEE Micro 28 (3) (2008) 42–53.

[22] A. Mukkara, N. Beckmann, D. Sanchez, Whirlpool: Improving dynamic cache management with static data classification, in: Proc. of ASPLOS'16, 2016, pp. 113–127.

[23] B. Gendron, T. G. Crainic, Parallel branch-and-branch algorithms: Survey and synthesis, Operations research 42 (6) (1994) 1042–1066.

[24] T. G. Crainic, B. Le Cun, C. Roucairol, Parallel branch-and-bound algorithms, Parallel combinatorial optimization 1 (2006) 1–28.

[25] S. Edelkamp, S. Schroedl, Heuristic Search: Theory and Applications, Morgan Kaufmann, 2012.

[26] J. Gmys, M. Mezmaz, N. Melab, D. Tuyttens, A GPU-based branch-and-bound algorithm using Integer-Vector-Matrix data structure, Par. Comp. 59 (2016) 119 – 139.

[27] T. Pessoa, J. Gmys, F. de Carvalho-Junior, N. Melab, D. Tuyttens, GPU-accelerated backtracking using cuda dynamic parallelism, Concurrency and Computation: Practice and Experience 30 (9) (2018) e4374.

[28] J. F. R. Herrera, J. M. G. Salmerón, E. M. T. Hendrix, R. Asenjo, L. G. Casado, On parallel branch and bound frameworks for global optimization, Journal of Global Optimization 69 (3) (2017) 547–560.

[29] L. Li, H. Liu, H. Wang, T. Liu, W. Li, A parallel algorithm for game tree search using gpgpu, IEEE Transactions on Parallel and Distributed Systems 26 (8) (2015) 2114–2127.

[30] T. Menouer, Solving combinatorial problems using a parallel framework, Journal of Parallel and Distributed Computing 112 (2018) 140 – 153.

[31] F. Galea, B. Le Cun, Bob++: a framework for exact combinatorial optimization methods on parallel machines, in: International Conference High Performance Computing & Simulation, 2007, pp. 779–785.

[32] H. Yun, R. Mancuso, Z. Wu, R. Pellizzoni, PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms, in: Proc. of RTAS'14, 2014, pp. 155–166.

[33] N. Beckmann, D. Sanchez, Modeling cache performance beyond LRU, in: Proc. of HPCA'16, 2016, pp. 225–236.

[34] IPyparallel. using IPython for parallel computing, `https://ipyparallel.readthedocs.io/`, accessed: 2019-03-19 (2018).

[35] Zeromq: An open-source universal messaging library, `https://zeromq.org/`, accessed: 2019-9-4.

[36] D. Beazley, Understanding the python GIL, in: In PyCON'10 Python Conference, 2010.

[37] JyNI jython native interface:compatibility/wish list, `https://jyni.org/#compatibility-wish-list`, accessed: 2019-11-21.

[38] BSC, Paraver: a flexible performance analysis tool, `https://tools.bsc.es/paraver`, accessed: 2019-03-19 (2018).

[39] J. C. Saez, A. Pousa, R. Rodríguez-Rodríguez, F. Castro, M. Prieto-Matias, PMCTrack: Delivering performance monitoring counter support to the OS scheduler, The Computer Journal 60 (1) (2017) 60–85.