Real-Time Systems manuscript No.
(will be inserted by the editor)

A Parallel Branch-and-Bound Algorithm to Compute a Tighter Tardiness Bound for Preemptive Global EDF

Mauro Leoncini · Manuela Montangero · Paolo Valente

Received: date / Accepted: date

Abstract In this paper we present a parallel exact algorithm to compute an upper bound to tardiness of preemptive Global EDF (G-EDF) schedulers, named *harmonic bound*, which has been proved to be up to 30% tighter than previously proposed bounds. Tightness is a crucial property of tardiness bounds: a too loose bound may cause a feasible soft real-time application to be mistakenly deemed unfeasible.

Unfortunately, no polynomial-time algorithm is known to date to compute the harmonic bound. Although there is no proof of hardness of any sort either, the complex formula of the bound apparently provides no hints to devise algorithms with sub-exponential worst-case cost.

In this paper we address this issue by proposing a parallel, exact, branch-and-bound algorithm to compute the harmonic bound, called *harm-BB*, which proves to be extremely fast in a large number of experiments. More specifically, we compare its execution times with those of existing polynomial-time algorithms for other known tardiness bounds on 630000 random task sets. *harm-BB* outperforms, or is comparable to, the competitor algorithms in all scenarios but the ones with the highest number of processors (7 and 8) and tasks ($\sim$50). In the latter scenarios *harm-BB* is indeed slower than the other algorithms; yet, it was still feasible, as it takes only about 2.8 s to compute the bound on a commodity Dual-Core CPU. Even better, we show that *harm-BB* has a high parallel efficiency, thus its execution time may be largely cut down on highly-parallel platforms.

Keywords Global EDF scheduler, Tardiness, Branch-and-Bound, Parallel algorithm

M. Leoncini, M. Montangero, P. Valente
Dipartimento di Scienze Fisiche, Informatiche e Matematiche
Università di Modena e Reggio Emilia
Via Campi 213/b
Modena, Italy
E-mail: name.surname@unimore.it

M. Leoncini
Istituto di Informatica e Telematica, CNR
Via Moruzzi 1
Pisa, Italy

1 Introduction

Complex time-sensitive applications play an important role in industry, business and daily life. Examples range from financial and IPTV services, to infotainment systems. The requirements of these applications are often less stringent than those of hard real-time (HRT) ones. In fact, these applications, modeled as sets of tasks that issue jobs to execute, usually tolerate deadline misses for some jobs, provided that some appropriate soft real-time (SRT) requirement is met. In many cases, it is sufficient that an application-specific maximum job-completion *tardiness* is guaranteed with respect to deadlines [15].

An important issue with these applications is that in most cases their computational demand can now be met only on multiprocessor platforms. On a multiprocessor, it is evidently more complex to schedule jobs so as to meet deadlines. We discuss existing solutions in the description of the related work (Section 1.2). Fortunately, as we highlight in that section, simple, global scheduling algorithms [7,27,10,11] are able to meet the requirements of any feasible SRT task set, provided that the latter tolerates a non-null tardiness with respect to deadlines. One of the lowest-overhead algorithms is Global Earliest Deadline First (G-EDF), which simply schedules jobs globally, in increasing deadline order.

On the downside, G-EDF may or may not succeed in meeting the requirements of a SRT application, depending on the actual maximum tardiness that it can guarantee to the tasks of the application. Moreover, maximum tardiness also affects the sizes of the buffers that may be used to conceal deadline misses. As a consequence, tardiness bounds for G-EDF [8,12,10,26], or, more precisely, the *tightness* of these bounds, play a critical role in determining the feasibility of applications scheduled with G-EDF. In fact, a too loose bound may cause a **feasible** application to be deemed **unfeasible**, or buffers to be uselessly too large. In this respect, the tardiness bound for G-EDF proposed in [26], named *harmonic bound* and valid for independent, implicit-deadline tasks on identical processors, proved to be up to 30% tighter than previously known bounds.

Moreover, as noted in [26], the harmonic bound follows from a general property shared by *every* work-conserving scheduling algorithm; as a consequence of this, it might help improve tardiness bounds and, in general, schedulability analysis also of other algorithms, techniques, and resources (other than CPU) [8,23,20,29]. One notable example, in this respect, is memory, that has clearly emerged as a critical shared resource in multi-cores, many-cores and GPUs processors [30,6].

The harmonic bound seems to suffer from a major drawback, i.e., a quite complex closed-form expression for which no polynomial-time algorithm is known to date for its computation, though a proof of hardness of some sort is not available either. These computational-cost issues are the motivation for the present contribution.

1.1 Contribution

We address the issue of efficiently computing the harmonic bound. A brute-force, exponential-time algorithm, here named as *harm-BF*, was proposed in [26] to com-

pute the harmonic bound. Briefly stated, *harm-BF* incurs a high computational cost because of the huge size of the space of task permutations that must be searched in order to optimize certain quantities (see Section 3). We have been unable to either prove any hardness result or provide a polynomial-time algorithm for the computation of the harmonic bound. However, in this paper we present a parallel algorithm based on the Branch and Bound technique, named *harm-BB*, which proves to be very effective in quickly pruning the permutation space to be searched.

In more details, the algorithm presented in this paper is an improved, parallel version of the preliminary, sequential algorithm that we introduced in a previous work [18]. The new algorithm exploits the fact that large portions of the above-mentioned task-permutation space can be searched independently from one another. This key property enables *harm-BB* to compute the harmonic bound through a number of parallel threads that in practice is only limited by the level of parallelism of the underlying hardware platform.

Before proceeding, we must also underline that [18] contained a few errors in the reported experimental results. More specifically, due to a subtle mistake in our C++ implementation of *harm-BB*, in [18] we reported unduly low *harm-BB* execution times for the most demanding scenarios. Here we report the correct execution times.

To assess the performance of *harm-BB*, we benchmarked both *harm-BB* and all the other polynomial algorithms over 630 groups of 1000 task sets each, with each group of task sets randomly generated according to one of 630 different combinations of: (1) number of processors, (2) utilization, and (3) period ranges. *harm-BB* proved to be faster than the competitor algorithms, or at least about as fast, in all scenarios, except for those with a number of processors equal to 7 and 8, with 8 being the maximum value investigated. Although in the latter scenarios *harm-BB* resulted significantly slower than the other algorithms[1], in nonetheless took no more 2.8 seconds to complete on an average Dual-Core CPU, even with the highest number of tasks (around 50) and processors (8).

In addition, as already stated, we prove that *harm-BB* has a high parallel efficiency, meaning that its execution time can be largely cut down on a highly-parallel platform.

1.2 Related work

One of three main scheduling paradigms is usually adopted to schedule jobs so as to meet deadlines: *global*, where any pending job can be executed on any available processor; *partitioned*, where the jobs of a given task can be executed only on a statically-selected processor; *semi-partitioned*, a variant of the partitioned scheme, where some tasks may *migrate* across processors. Optimal, global multiprocessor scheduling algorithms have been devised by [1,2,22,24]. They guarantee all deadlines to feasible task sets (i.e., task sets with a total utilization not higher than the total platform capacity), but are relatively complex, and may incur non-trivial overheads [4,5]. On the opposite end, partitioned and semi-partitioned schemes are much simpler, and incur

[1] These are the scenarios for which wrong execution times, in the order of just milliseconds, were reported for the sequential version of *harm-BB* in [18].

significantly lower overheads. However, these schemes are not optimal. In this respect, a simple semi-partitioned approach that combines many different techniques has recently been proposed [5]. The authors proved that such an approach is able to schedule all HRT task sets, among those considered in their experiments, with a total utilization of up to 99% of the system capacity (and occasionally even more).

However, if a feasible SRT task set tolerates a non-null tardiness with respect to deadlines, then even very simple global schedulers [7, 27, 10, 11] exist which are able to meet the requirements of the task set, independently of its total utilization. One of the simplest algorithms owning this feature is G-EDF, which has been proven to guarantee bounded tardiness for every task set with a total utilization not exceeding the total capacity of the system [7, 27]. G-EDF is even simpler than the solution proposed in [5], as the latter incorporates many non trivial features, which include: a policy to allocate tasks on processors, a bandwidth server, an instance of (uni-processor) EDF running for each processor, as well as a few other heuristic criteria. As for feasibility on real systems, G-EDF is strongly believed to be an effective and low-overhead solution for SRT tasks.

In addition, according to [19] and [20], G-EDF seems effective also for new parallel task models. Special variants targeted at these more complex task models have been proposed by [29]. In this respect, the state-of-the-art results with a semi-partitioned approach [21] are still far from the optimal results achieved in [5] for the classical independent, implicit-deadline sporadic (or periodic) real-time tasks.

Three main tardiness bounds for G-EDF have been devised before the harmonic bound was proposed. In its tightest version, the first such bound appeared in [8]. The second bound, again in its tightest form, was proposed by [12] as an improvement (which was already discussed in [8]) of *compliant-vector analysis* (*CVA*). The third bound was obtained by [10] through an alternative improvement on *CVA*, named *PP Uniform Reduction Law*.

To compute all the above mentioned bounds, polynomial-time algorithms are available which are exceedingly faster than *harm-BF*. However, in the conference version of this paper we observed that *harm-BF* seemed amenable to highly efficient parallel implementation on modern multi- and many-core architectures. Here we provide one such implementation, based on the Branch and Bound approach, that proves the correctness of our observation.

Although many good works are present in the literature on parallel Branch-and-Bound, some of which also provide usable frameworks (see, e.g., [9]), turning a specific algorithm into highly efficient parallel code always presents peculiarities and challenges, as correctly noted in [16]. In particular, the issue of interprocess communication and synchronization, mainly resulting from the need of coordinating the exploration of the huge search space, is one that might frustrate the gain due to the possibly much higher computing power available.

To cope with the above problem, an extreme solution is to give up doing space partitioning at all. This is suggested, for instance, in the first approach presented in the above mentioned paper [16]. There, the speed-up over sequential implementations is expected from at least one process to start space exploration equipped with a good approximation of the optimum (solution) value, which would lead to a great amount of pruning. Of course, success here highly depends on the application domains and, in

particular, on the availability of probabilistic information about the optimum value. We also note that avoiding (or highly constraining) space partitioning also reduces the possible arising of known anomalies (see, e.g., [17]).

In our case, we can limit interprocess communication and other more subtle implementation problems due to the particular nature of the space being explored.

Roadmap

In Section 2 we describe the system model. In Section 3 we explain the harmonic bound in details and discuss the brute force algorithm. In Section 4, we present the branch and bound algorithm, and, in Section 5, we describe the experimental setting and discuss the obtained results.

2 Task and service model

In this section we introduce the model adopted in the paper and the notation (summarized in Table 1).

Table 1: Main notations used throughout the paper.

Model	
τ, N	Set and number of tasks
$\tau_i \in \tau$	i-th task in τ
C_i	Worst-case computation time of the jobs of task τ_i
T_i	Period/Minimum inter-arrival time of the jobs of task τ_i
M	Number of symmetric processors
$U_i = \frac{C_i}{T_i} \leq 1$	Utilization of task τ_i
$U_{sum} = \sum_{i \in \tau} U_i \leq M$	Total utilization of the task set
$U = \lceil U_{sum} \rceil - 1$	
Harmonic Bound and B&B Algorithm	
G-long permutation	Permutation with G elements
$\Pi(\tau, G)$	Set of all G-long permutations of tasks in τ
$c\Pi(\pi_j, G, h)$	Set of G-long tail-constrained permutations with tail π_j and head length h
$S \in \{\Gamma, \Omega\}$	Depending on which function *harm-BB* is computing
S_{LB}	Value of current best solution
$S_{UP}(\pi_j)$	Upper Bound for $S(\pi_i)$, with $\pi_i \in c\Pi(\pi_j, G, h)$

2.1 Task model

We consider a set τ of N asynchronous *tasks* τ_1, τ_2, ..., τ_N, each one consisting of an infinite sequence of *jobs* to execute. The j-th job of τ_i, denoted by J_i^j, is characterized by a 4-tuple, $(r_i^j, c_i^j, f_i^j, d_i^j)$ where:

- r_i^j is the *release* (or *arrival*) time of J_i^j;
- c_i^j is the *computation time*, namely the time needed to execute J_i^j on a unit-speed processor;
- f_i^j is the job *finish time*;
- d_i^j is an *absolute deadline*, i.e., the time within which the job should be finished;

A fifth important parameter is the job *tardiness*, t_i^j, which is defined in terms of deadline and finishing time, i.e.:

$$t_i^j = \max(0, f_i^j - d_i^j)$$

No offset is specified for the release time of the first job of any task.

Let $C_i = \sup_j c_i^j$ and $T_i = \inf_j \left(r_i^{j+1} - r_i^j \right)$ be the *maximum computation time* and *minimum inter-arrival time* of the jobs of task τ_i, respectively. For all j it clearly holds $r_i^{j+1} \geq r_i^j + T_i$. The pair (C_i, T_i) somehow characterizes task τ_i. In particular, for any $j \geq 1$ the absolute deadlines d_i^j are given implicitly and set to $r_i^j + T_i$.

We define as *tardiness of a task* the maximum tardiness experienced by any of its jobs, and for each task τ_i we define its *utilization* as $U_i \equiv \frac{C_i}{T_i} \leq 1$. Finally, we denote by U_{sum} the total utilization $\sum_{\tau_i \in \tau} U_i$ of the task set, and by U the quantity $\lceil U_{sum} \rceil - 1$.

2.2 Service model

The harmonic bound holds for a task set τ, defined as in the previous subsection, under the following conditions:

- jobs are executed on a multiprocessor with M identical unit-speed processors;
- M satisfies the inequalities $M < N$ and $U_{sum} \leq M$;
- jobs are scheduled according to the global and preemptive EDF (G-EDF) policy, meaning that 1) each time a processor becomes idle, the pending (and not yet executing) job with the earliest deadline is dispatched on it, 2) when a job J arrives whose deadline comes before that of at least one running job, then the job in execution with latest deadline is *preempted* (i.e., suspended) in favor of J, with ties broken arbitrarily.

3 The harmonic bound

In this section, we first recall the formula of the harmonic bound, then we provide a short description and analysis of the brute-force algorithm, *harm-BF*, which can

be used to compute the bound. A detailed description of the harmonic bound can be found in [26].

The formula of the harmonic bound is rather long and complex and unfortunately we cannot offer any easy intuitions to help understand it. However, for the purposes of the present paper, what is important is to catch on the "structure" of (the formula of) the bound. The latter can be expressed in terms of two maximization functions on task sets, which we will denote as Γ and Ω. In turn, to closely reflect actual *harm-BB* computations, we define Γ and Ω via two more general functions, $\Gamma(\cdot)$ and $\Omega(\cdot)$, respectively, that apply to task permutations.

We start with two definitions that will be used through all the paper. Recall that, by definition, $U = \lceil U_{sum} \rceil - 1$ is a natural number.

Definition 1 (G-long permutation and set) Given the set of tasks τ and an integer $G \leq N(= |\tau|)$, we say that a permutation $(\tau_{i_1}, \ldots, \tau_{i_G})$ of elements in τ with cardinality $G \leq N$ is a G-long permutation. Moreover, we write $\Pi(\tau, G)$ to denote the set of all $\binom{N}{G}$ G-long permutations.

Definition 2 (M-contribution) Given an integer $G \leq U$ and a G-long permutation $\pi_i \in \Pi(\tau, G)$, for any $g \leq G$ we define

$$M_g^{\pi_i} \equiv M - \sum_{v=1}^{g-1} U_{i_v}. \tag{1}$$

$M_g^{\pi_i}$ can be regarded as the *residual* utilization available after the first $g-1$ tasks of the permutation. We are now able to introduce our first auxiliary function $\Gamma(\cdot)$.

Function $\Gamma(\cdot)$. Given a U-long permutation π_i, we define

$$\Gamma(\pi_i) \equiv \Gamma((\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_U})) \equiv M \cdot \sum_{g=1}^{U} \frac{C_{i_g}}{M_g^{\pi_i}}. \tag{2}$$

Note that the right-hand side of Equation (2) contains a summation of U fractions, one for each task in the permutation. Given the i^{th} fraction we have that: (1) the numerator is the task completion time of the i^{th} task in the permutation, (2) the denominator is a M-contribution that depends on the utilizations of the tasks with indices smaller than i in the permutation.

We now define $\Gamma(\tau)$ simply as the maximum, over all possible U-long permutations π_i in τ, of $\Gamma(\pi_i)$, *i.e.*,

$$\Gamma(\tau) = \max_{\pi_i \in \Pi(\tau, U)} \Gamma(\pi_i). \tag{3}$$

The formula for our second auxiliary function, $\Omega(\cdot)$, is slightly more complex and makes use of Γ.

Function $\Omega(\cdot)$. Given a G-long permutation $\pi_i \in \Pi(\tau, G)$, with $1 \leq G \leq U$, we define

$$\Omega(\pi_i) \equiv \Omega((\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_G})) \equiv \frac{1}{M} \cdot M_{G+1}^{\pi_i} \cdot \left(\underbrace{\Gamma(\tau) \cdot \sum_{g=1}^{G} \frac{U_{i_g}}{M_g^{\pi_i} M_{g+1}^{\pi_i}} + \sum_{g=1}^{G} \frac{C_{i_g}}{M_g^{\pi_i}}}_{\omega(\pi_i)} \right). \quad (4)$$

Notice that $\omega(\pi_i)$ contains two summations with G terms each, one for each task in the permutation, and that such terms are fractions. Given the i^{th} fraction of one of the two summation, we have that: (1) numerators are either task utilizations or completion times of the i^{th} task in the permutation, (2) denominators are defined in terms of M-contributions. For the summation on the left, the involved M-contributions depend on the utilizations of all tasks in positions 1 through i. For the summation on the right, only utilizations of tasks up to position $i-1$ are involved.

We define $\Omega(\tau)$ as the maximum of $\Omega(\pi_i)$ over all possible G-long permutations π_i in τ, for $1 \leq G \leq U$, *i.e.*,

$$\Omega(\tau) \equiv \max_{\substack{\pi_i \in \Pi(\tau, G) \\ 1 \leq G \leq U}} \Omega(\pi_i). \quad (5)$$

A few observations are in order. First, for simplicity we will write Γ and Ω instead of $\Gamma(\tau)$ and $\Omega(\tau)$, since τ is the (fixed) set of tasks. Second, even if Ω does depend on Γ, the permutations that maximize the two functions are not necessarily the same. Moreover, while Γ is computed by maximizing over the permutations of exact length U, the definition of Ω takes all permutations of size at most U into consideration.

The Harmonic Bound. We are now ready to state the harmonic bound for the set of tasks τ as a function of Ω (and hence also of Γ).

Theorem 1 (Harmonic Bound) *For every job J_i^j of each task τ_i we have*

$$f_i^j - d_i^j \leq \Omega + \frac{M-1}{M} C_i. \quad (6)$$

As proven by Lemma 2 in [26], the right-hand side (RHS) of (6) is not negative, and is therefore a tardiness bound (recall that tardiness is a non-negative quantity).

3.1 *harm-BF*: a brute-force approach

The time-consuming part in computing the harmonic bound is of course the computation of Γ and Ω. Given a task set τ, the brute force approach *harm-BF* proceeds as follows:

1. **Compute** Γ. Generate all the possible U-long permutations $\pi_i \in \Pi(\tau, U)$, and compute the value $\Gamma(\pi_i)$ defined in (2) for each such permutation. Then, according to (3), take the maximum among these values to get Γ.
2. **Compute** Ω. Generate all the possible G-long permutations $\pi_i \in \Pi(\tau, G)$, for all $G \in \{1, \ldots, U\}$, and compute the value $\Omega(\pi_i)$ in (4) for each such permutation, using the value of Γ computed in step (1). Then, according to (5), take the maximum among these values.
3. **Compute the RHS of** (6) **for every task in** τ, using the value of Ω computed in step (2).

The time complexity of *harm-BF* is clearly factorial.

4 *harm-BB*: a Branch-and-Bound approach

In this section we present our parallel algorithm *harm-BB* which parallelizes a Branch-and-Bound approach to compute the harmonic bound. In particular, functions Γ and Ω are computed independently, Γ first, and in both cases a parallel Branch-and-Bound algorithm is used. By default, *harm-BB* generates a number of parallel execution threads equal to the number of logical processors that run the algorithm. Alternatively, the number of parallel threads can be given as an input parameter. In this way, by setting this number to one, we can simulate a sequential execution of the algorithm.

As the high level approach in computing the two functions is often analogous, when appropriate, we will first describe the algorithm referring to any of the two functions using a generic alias $S \in \{\Gamma, \Omega\}$, and then giving specific indications for each function when needed.

The rest of this long section is organized as follows: we will start by recalling the basic concepts of the Branch-and-Bound technique, and then we will describe how to apply and parallelize the technique to the special case of computing the harmonic bound. Pseudo-code for the bound computation is given in Figures 3 and Figure 6, and referenced throughout the text when needed.

4.1 Branch-and-Bound basics

Here, we give a succinct high-level description of the Branch-and-Bound approach [14] for the solution of combinatorial optimization problems. Authors familiar with such technique may directly proceed to Section 4.2.

We describe the approach for maximization problems, like the ones involved in the computation of the harmonic bound. For minimization problems dual considerations apply. A brute force algorithm for a maximization problem explores the search space (*i.e.*, the space of all the feasible solutions to the problem) by computing the objective function at each point of the space and returning the solution with maximum value. The Branch-and-Bound approach also explores the search space but with the aim at reducing the number of evaluations. As the name suggests, the two main operations performed by a Branch-and-Bound algorithm are *branching*, which amounts

to partitioning a subspace of the search space into smaller sets, and *bounding*, which is the computation of an upper bound to the values of all the solutions belonging to a particular subspace. Initially, the subspace under investigation is the whole search space A. At the generic step, the algorithm considers one particular subspace $A' \subseteq A$ and the value x of the best solution found so far; the algorithm tries to compute an upper bound b to the value of any solution in A'. If $b \leq x$, then the subspace is abandoned, since it cannot possibly contain a solution better than the current one. This operation is called *pruning*. If instead $b > x$ holds, then A' deserves further investigations. In this case, the algorithm further partitions the subspace[2] and marks the generated spaces for future analysis. Clearly, a subspace that includes one single solution cannot be further partitioned. In this case the algorithm explicitly computes the value of the corresponding solution, which possibly replaces the best current one. The algorithm stops when there are no more subsets to bound; it then returns the best current solution found as the value of the optimal solution to the problem.

The key to efficiency for Branch-and-Bound algorithms is clearly the possibility to prune large portions of the search space using a relatively small number of bounding operations. When this happens, we observe a significant decrease in the running time (with respect to brute force). Obviously, the Branch-and-Bound approach is not the magic wand to cope with high-complexity problems; indeed, in the worst case all (or almost all) feasible solutions must be evaluated to locate an optimal one and thus the worst-case running time is asymptotic to that of brute force.

Execution tree. The execution of a Branch-and-Bound algorithm can be represented by a (dynamically growing) rooted tree T, in which each node corresponds to a subset of the search space. The *level* of a node $x \in T$ is the number of edges of the unique path in T joining x to the root. The root itself is the only node at level 0. Initially, T includes only the root r which corresponds to the whole search space.

In general, a node x at level ℓ represents a subset of the search space that we denote with V_x. The set V_x results from the partition of (the subspace associated to) its parent node (at level $\ell - 1$). Nodes that are (still) not partitioned are *leaves* of T.

Each leaf of the tree is either *dead* or *alive*. Dead leaves correspond to pruned subsets, while alive leaves correspond to those subsets that did not undergo bounding yet. When an alive leaf x undergoes branching, a set of children is generated and added to the tree, each one corresponding to a subset of V_x.

An example is depicted in Figure 1.

In the following of this section, we will concentrate on the computation of functions Γ and Ω. We will define the search spaces and subsets V_xs for the two functions. Then, we will describe how to compute a first best current solution, how to branch and how to bound. Finally, we will prove the correctness of the computed upper bounds.

[2] The way this partition is made, as well as the number of generated subspaces, is clearly problem dependent.

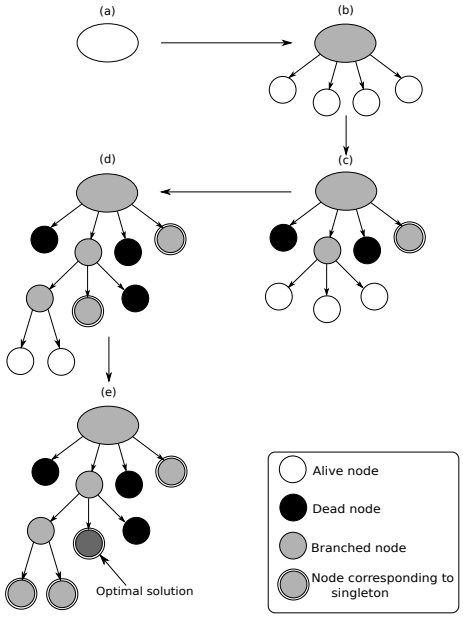


Fig. 1: Example of execution tree for a Branch-and-Bound algorithm. Initially, the root of the tree is the only alive node. At each branching operation, a set of sibling is added as children of a former alive node. Each bounding operation determines if one alive node is to be branched or declared dead. For a leave corresponding to a singleton, the actual value of the function is computed with input that singleton, and only the best solution is retained.

4.2 Search spaces and execution trees

For both function Γ and function Ω the search space is a set of permutations of tasks. In particular,

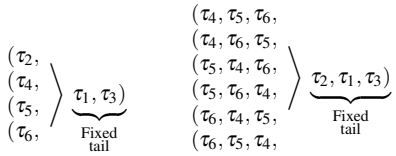
- Function Γ: the search space is the set $\Pi(\tau, U)$, i.e., the set of all U-long permutations;
- Function Ω: the search space is the set $\cup_{1 \leq G \leq U} \Pi(\tau, G)$, *i.e.*, the set of all permutations with length ranging from 1 to U.

The execution of *harm-BB* considers subsets of feasible solutions in the search space, each corresponding to one node of the execution tree, i.e., the tree that represents the algorithm execution. To describe such subsets we need to introduce the concepts of *Head* and *Tail* of a permutation and of *tail-constrained permutations*.

Definition 3 (Head and Tail) Given three natural numbers h, t and G, such that $h + t = G \leq N$, and a G-long permutation

$$\pi_i = (\tau_{i_1}, \ldots, \tau_{i_G}) \in \Pi(\tau, G),$$

we define:



(a) $c\Pi((\tau_1,\tau_3),3)$ (b) $c\Pi((\tau_2,\tau_1,\tau_3),6)$

Fig. 2: Example of tail constrained permutations. Given the set of tasks $\tau = \{\tau_1,\tau_2,\tau_3,\tau_4,\tau_5,\tau_6\}$: (a) set of tail constrained permutations with $G = 3$ and $\pi_j = (\tau_1,\tau_3)$; (b) set of tail constrained permutations with $G = 6$ and $\pi_j = (\tau_2,\tau_1,\tau_3)$.

- The **head** of π_i of length h as the permutation

$$H(\pi_i,h) = (\tau_{i_1},\ldots,\tau_{i_h}),$$

  i.e., the first h elements of permutation π_i;
- The **tail** of π_i of length t as the permutation

$$T(\pi_i,t) = (\tau_{i_{G-t+1}},\ldots,\tau_{i_G}),$$

  i.e., the last t elements of permutation π_i.

Definition 4 (Tail-constrained permutations) Given a natural number G, such that $0 \leq G \leq N$, and a t-long permutation $\pi_j = (\tau_{j_1},\ldots,\tau_{j_t}) \in \Pi(\tau,t)$, for some $t \leq G$, we define the set of *tail-constrained* permutations as:

$$c\Pi(\pi_j,G) \equiv \{\pi_i \in \Pi(\tau,G) \mid T(\pi_i,t) = \pi_j\}, \tag{7}$$

i.e., all G-long permutations whose tail of length $t = |\pi_j|$ is equal to the given permutation π_j (See Figure 2 for a couple of examples).

We can now be more precise about the permutation spaces associated to the nodes of the Branch-and-Bound tree resulting from the execution of *harm-BB*. Intuitively, each node of the tree corresponds to a set of permutations sharing the same tail.

Function Γ. A node at level $t \leq U$ corresponds to a set of U-long tail-constrained permutations $c\Pi(\pi_j,U)$, where each permutation has tail $\pi_j \in \Pi(\tau,t)$ of length t.

For $t = 0$, the root corresponds to the set of U-long tail-constrained permutations with degenerate empty tail. Leaves that can not be further branched, correspond to singletons, that is, sets of U-long tail-constrained permutations with degenerate empty heads.

Referring to the example in Figure 2, the set in (a) corresponds to a node at level $t = 2$ when $U = 3$, while set in (b) to a node at level $t = 3$ when $U = 6$.

Function Ω. To compute Ω, *harm-BB* builds a forest of execution trees, instead of just one single tree. In particular, there are U execution trees, one for each integer value $G \in \{1,\ldots,U\}$. Each tree accounts for permutations of fixed length G and is analogous to the execution tree for Γ, with the sole difference of the length of the permutations. For $t = 0$ and $G \in \{1,\ldots,U\}$, the root corresponds to the set of G-long tail-constrained permutations with degenerate empty tail.

Trees are built one after the other, in decreasing order with respect to the length of the permutations.

Execution trees are computed by leveraging parallelism. Indeed, the computation on distinct subtrees involves distinct subsets of the search space, and hence it is very natural to assign distinct subtrees to distinct parallel thread for their computation. In particular, *harm-BB* assigns to idle threads nodes at level one in the execution tree which have not been taken into consideration yet, and each thread computes the whole subtree rooted at such node (Lines 20-25 of code in Figure 6).

The choice of assigning to threads only nodes at level one in the execution tree is motivated by the fact that blocking and waking up threads takes time, and this can reach microseconds. To get a high parallel efficiency, threads must execute computations that last more than this overhead time. Therefore, each thread deals with large subtrees.

4.3 Best current solution

harm-BB keeps and updates the value of the best current solution found until that moment, denoted with S_{BC}. During the algorithm execution, when *harm-BB* encounters a leaf that corresponds to a singleton $\{\pi_i\}$, it computes $S(\pi_i)$ and compares this value with that of the current best solution S_{BC}. If $S_{BC} < S(\pi_i)$, then π_i becomes the new (current) best solution and *harm-BB* updates the value of S_{BC} by setting $S_{BC} = S(\pi_i)$.

Observe that S_{BC} is a global variable and thus operating on it might generate conflicts among threads. To circumvent this, we could force a thread lock the variable any time it needs to access the variable. Unfortunately, such an easy solution is likely to result in a poorly efficient parallel implementation, as threads might waste time waiting for the variable to be unlocked. To minimize locking on S_{BC}, we observe that, being S_{BC} a scalar, we can (safely) assume that any update of its value is atomic. This implies that, if a thread tries to read S_{BC} while another thread is updating it, then the first thread sees either the previous or the new value, but not an inconsistent value. Therefore, *harm-BB* makes a thread read S_{BC} without taking the lock, but requires a lock on S_{BC} if the thread has to update its value. After locking S_{BC}, the thread has to check again if it really has to update the value of the variable. Indeed, it might happen that the value of S_{BC} has been updated by another thread in between the read operation and the decision to update the value. Therefore, the update occurs only if the locked value of S_{BC} is smaller than the value $S(\pi_i)$ computed by the thread (Lines 4-11 of code in Figure 6) .

4.4 Initial solutions

The initial S_{BC} value is set by taking the highest among two quantities, corresponding to specific solutions that are deemed to be not too distant from the optimal value. We denote such solutions τ_C and τ_U, respectively. These are composed of the U-long permutations with largest C_is (resp. smallest U_is) values, sorted in increasing (resp. decreasing) order. The intuition is that we try to maximize the fractions in the expressions in (2) and (4): by taking the largest C_is we tend to maximize the numerators; by taking the smallest U_is, we tend to minimize the M-contributions at the denominators.

Then, at the beginning, we set (Lines 1-5 of code in Figure 3)

$$S_{BC} = \max\{S(\tau_C), S(\tau_U)\}.$$

For what concerns the function Ω, Ω_{BC} is the initial solution for the execution tree with U-long permutations. For the other trees, we also take the best solution computed by previous execution trees into consideration.

```
COMPUTEΩ(τ, U)
 1  // Compute initial values for Γ and Ω
 2  Compute τ_U
 3  Compute τ_C
 4  Γ_BC := max{Γ(τ_U), Γ(τ_C)} // Global variable
 5  Ω_BC := max{Ω(τ_U), Ω(τ_C)} // Global variable
 6  // Generate parallel execution threads
 7  GENERATEPARALLELTHREADS(num_logical_CPUs)
 8  // Compute Γ;
 9  // U is the length of the permutation, () is the empty tail
10  Γ := EXECUTIONTREEΓ(U, ()) // Uses global variable Γ_BC
11  // Compute Ω
12  Ω := 0
13  // start one execution tree for each G
14  for G = U downto 1
15      // G is the length of the permutation, () is the empty tail
16      Ω := max{Ω_BC, EXECUTIONTREEΩ(G, ())} // Uses global variable Ω_BC
17  return Ω
```

Fig. 3: Pseudo-code of main thread for the computation of function Γ given set of tasks τ and $U = \lceil U_{sum} \rceil$, where U_{sum} is the total utilization. GENERATEPARALLELTHREADS($num\_logical\_CPUs$) generates a number of parallel threads equal to the number of logical CPUs; EXECUTIONTREEΓ and EXECUTIONTREEΩ are recursive functions, whose pseudo-code can be found in Figure 6, that generate and explore the Branch-and-Bound execution trees for the two functions and use the global variables S_{BC}.

4.5 Branching

Branching is the operation executed at a given alive node x of the execution tree (Lines 14-29 of code in Figure 6) to add new leaves. The set V_x of solutions associated to x is a set of constraint permutations and, intuitively, branching is done by simply increasing the length of the tail by one, in all possible ways (hence partitioning the set V_x). The branching strategy is the same for the execution tree of Γ and for each tree of Ω.

We give a formal general description for a node corresponding to a generic subset of G-long permutations, with $G \in \{1,\ldots,U\}$, and tail π_j.

Consider a node x at level $t < G$ in the tree, corresponding to the set of G-long permutations $c\Pi(\pi_j, G)$. For a given t-long tail $\pi_j = (\tau_{j_1},\ldots,\tau_{j_t})$, let $\tau^{(j)}$ be the set of task in τ excluding those in the tail π_j; *i.e.*, $\tau^{(j)} = \tau \setminus \{\tau_{j_1},\ldots,\tau_{j_t}\}$. The branching procedure generates a new leaf in the tree for each task $\tau' \in \tau^{(j)}$, *i.e.*, that is not already in the tail. Each leaf is set as child of node x and corresponds to the subset of G-long permutations with tail composed by the task τ' followed by the tasks in π_j, i.e., the set $c\Pi((\tau', \tau_{j_1},\ldots,\tau_{j_t}), G)$.

Observe that branching at a single node can be done in polynomial time, for both functions.

Example 1 Figure 4 shows an example of a portion of a complete execution tree, with the indication of the corresponding V_x for each node x. In this example, the task set is $\tau = \{\tau_1, \tau_2, \tau_3, \tau_4\}$. The root node corresponds to the set of all 3-long permutations on the task set τ. The node at level one corresponds to the subset of 3-long permutations with tail τ_1, and all possible other alternatives in the two positions of the head. Its children, at level two, are obtained by increasing the tail by one task, choosing among all possible tasks in τ, except τ_1 (which is already in the tail). The chosen task is the label of the outgoing edge. Each child node corresponds to a set of 3-long permutations with two tasks in the tail, and the constraint that the last one is τ_1. This leads to have three children at level two of the tree, corresponding to three sets of 3-long permutations with tails $(\tau_2, \tau_1), (\tau_3, \tau_1)$ and (τ_4, τ_1). Finally, at level three there are nodes with a tail that is composed by three tasks, i.e. they correspond to singletons and contain exactly one feasible solution. In this case the tail has the length of the permutation and the head is empty. These nodes can not be further branched.

After being added to the tree, the new leaves have to undergo bounding. *harm-BB* orders the sibling leaves in decreasing order of completion time of the new task added to the tail during the branching operation. This will be the order in which such nodes will be considered for bounding. Referring again to Figure 4, if we assume that $C_1 \geq C_2 \geq C_3 \geq C_4$, sibling nodes are evaluated form left to right. This heuristic is intended to examine the subset that might contain a solution with larger value first and, thus, prune the subtrees of the siblings earlier (using completion times empirically proved to be more effective than using utilization for the same purpose).

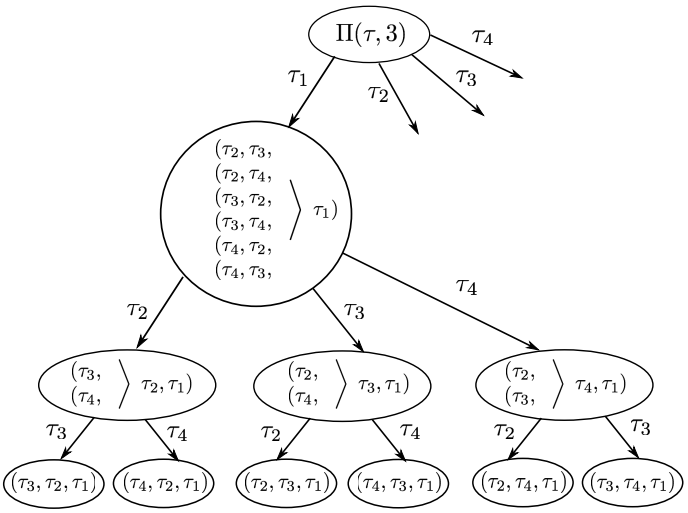


Fig. 4: Figure relative to example 1: portion of complete execution tree. Inside each node of the tree there is the indication of the corresponding subset of the set of 3-long permutations over the set $\tau = \{\tau_1, \tau_2, \tau_3, \tau_4\}$. The root corresponds to the whole set of permutations $\Pi(\tau, 3)$. Labels on edges indicate the task that has been selected to extend the tail of the father node to derive the children nodes.

4.6 Bounding

Bounding is the operation that allows *harm-BB* to decide whether a leaf x is dead or alive. Given the corresponding set V_x of G-long tail constraint permutations with tail π_j, *harm-BB* computes an upper bound $S_{UP}(\pi_j)$ for the values of the solutions in the set V_x; *i.e.*, we have

$$S(\pi_i) \leq S_{UP}(\pi_j) \tag{8}$$

for each G-long permutation π_i with tail π_j.

If $S_{UP}(\pi_j) \leq S_{BC}$ (the value of the best current available solution), then the whole set of permutations in V_x is pruned from the search space, *i.e.*, x is declared dead. Otherwise, x is declared alive and will undergo branching (Line 13 of code in Figure 6).

The value $S_{UP}(\pi_j)$ is computed by substituting specific values in the function $S(\cdot)$ in order to be sure that (8) holds. It is straightforward to obtain an upper bound if all values C_{i_g}s and U_{i_g}s that are relative to tasks in the head of π_i are substituted with the maximum C_{max} and maximum U_{max} values. Indeed, C_is and U_is give a positive contribution in the numerators, while the U_is give a negative contribution in the denominators.

However, with a bit more care we obtain a tighter bound, which should allow *harm-BB* to declare dead a larger number of nodes with respect to a looser one.

The idea is to use not only C_{max} and U_{max}, but the h largest C_{i_g}s and the h largest U_{i_g}s. In particular, all values C_{i_g}s and U_{i_g}s that are relative to tasks in the head of π_i

are substituted with the h largest C_is (in increasing order) and the h largest U_is (in decreasing order) among all tasks in τ, excluding those in the fixed.

To formalize this idea we define h virtual tasks having the desired completion times and utilizations and then we give a complete example.

Definition 5 (Virtual tasks) Given a $G \leq U$ and a t-long tail $\pi_j \in \Pi(\tau,t)$, with $t \leq G$, the set $\tau^{(j)} \subseteq \tau$ is the set of tasks in τ that do not belong to the tail π_j. We define $h = G - t$ *virtual* tasks $\hat{\tau}_1, \ldots, \hat{\tau}_h$ in the following way:

1. Order the U_is of the tasks in $\tau^{(j)}$ in decreasing order, obtain the sequence $U_{r_1} \geq U_{r_2} \geq \ldots \geq U_{r_{|\tau|-t}}$ and retain the largest h U_is:
$$U_{r_1} \geq U_{r_2} \geq \ldots \geq U_{r_h}. \tag{9}$$
2. Order the C_is of the tasks in $\tau^{(j)}$ in increasing order, obtain the sequence $C_{k_1} \leq C_{k_2} \leq \ldots \leq C_{k_{|\tau|-t}}$ and retain the largest h C_is (starting with the index $(|\tau|-t) - h+1 = |\tau| - G+1$):
$$C_{k_{|\tau|-G+1}} \leq C_{k_2} \leq \ldots \leq C_{k_{|\tau|-t}}. \tag{10}$$
3. For $y = 1, \ldots, h$, the virtual task $\hat{\tau}_y$ is characterized by the following values of utilization $\hat{U}_y$ and completion time $\hat{C}_y$:
   - $\hat{U}_y = U_{r_y}$; i.e., the y^{th} U_i starting from the right of the ordered sequence in (9);
   - $\hat{C}_y = C_{k_{|\tau|-G+y}}$; i.e., the y^{th} C_i starting from the right of the ordered sequence in (10).

In other words, completion times and utilizations of the virtual tasks coincides with the h largest in $\tau^{(j)}$. Observe that virtual tasks exhibits increasing completion times and decreasing utilizations. Moreover, notice that virtual tasks may or may not actually exist in τ. This is not an issue, as these tasks are used to compute the upper bound to the values of feasible solutions in a given subset of task permutations, and not the value of one feasible solution.

Example 2 Figure 5 shows an example of how to define virtual tasks. (a) The task set is $\tau = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7\}$, we are considering 5-long permutations with a 3-long tail equal to $\pi_j = (\tau_2, \tau_5, \tau_4)$ (hence, the head length is 2). The set of tasks that are not in the tail, and that are used to define the virtual tasks, is $\tau^{(j)} = \{\tau_1, \tau_3, \tau_6, \tau_7\}$. (b) Steps (1) and (2) of Definition 5 are performed: utilizations of tasks in $\tau^{(j)}$ are ordered in decreasing order; completion times of tasks in $\tau^{(j)}$ are ordered in increasing order; the boxed values are the $h = 2$ largest values selected in the ordered sequences (9) and (10). (c.1) Two virtual tasks are defined: $\hat{\tau}_1$ has utilization equal to the first (leftmost) boxed value and completion time equal to the first (leftmost) boxed value; $\hat{\tau}_2$ has utilization equal to the second (from the left) boxed value and completion time equal to the second (from the left) boxed value. Observe that $\hat{\tau}_1 = \tau_3$, while $\hat{\tau}_2$ is a task that does not exist in τ.

We are now ready to define the permutation that is given as input to the functions that are used to compute the upper bounds.

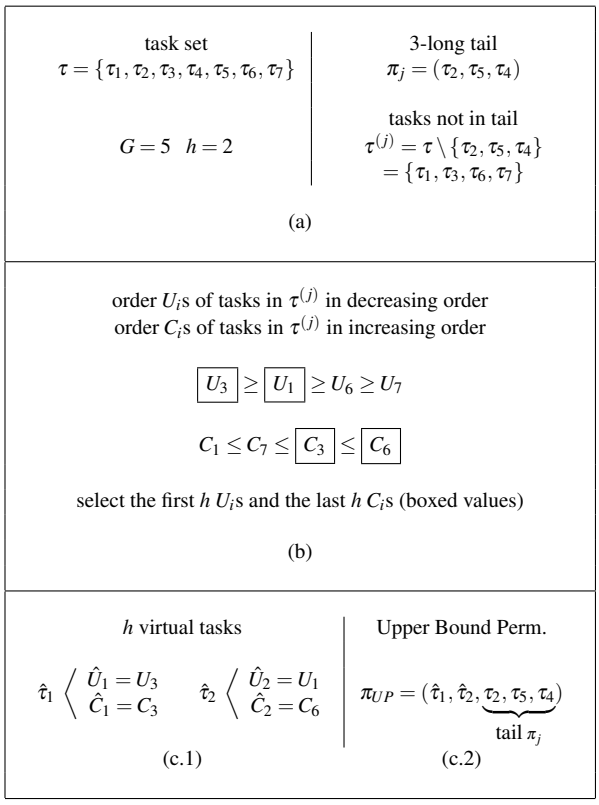


Fig. 5: Figure relative to Examples 2 and 3: computation of virtual tasks and upper bound permutation. (a) Input values for the examples. (b) Steps (1) and (2) of Definition 5, which determine the utilizations and the completion times of the virtual tasks. (c.1) Virtual tasks determined according to step (3) of Definition 5. (c.2) Upper Bound permutation determined according to Definition 6

Definition 6 (Upper Bound permutation for tail π_j) The upper bound permutation for the G-long tail constrained permutations with t-long tail π_j is denoted with π_{UP} and is the sequence of G tasks obtained by concatenating the sequence of virtual tasks $(\hat{\tau}_1, \ldots, \hat{\tau}_h)$ with the tasks in the tail π_j.

Example 3 Figure 5 (c.2) shows the resulting 5-long upper bound permutation π_{UP} relative to Example 2: the head is composed by the two virtual tasks $\hat{\tau}_1$ and $\hat{\tau}_2$, while the tail by the tasks in the given tail π_j.

4.6.1 Computing upper bounds

In this section we define upper bounds to the values of the solutions in the set V_x corresponding to node x of the execution tree. These bounds are obtained by calculating

the value of specific functions (identical, or very similar, to Γ and Ω) on input the upper bound permutation π_{UP} defined in the previous section.

We give details for the two functions separately.

Upper Bound to function Γ. Given a t-long tail permutation $\pi_j \in \Pi(\tau,t)$ and a U-long tail-constrained permutation π_i with tail π_j, we restate the definition of $\Gamma(\pi_i)$ given in equation (2) by splitting the summation in two parts: one concerning tasks in the head, the other concerning tasks in the tail:

$$\Gamma(\pi_i) = M \cdot \left(\underbrace{\sum_{g=1}^{h} \frac{C_{i_g}}{M_g^{\pi_i}}}_{\substack{\text{head addends} \\ (a)}} + \underbrace{\sum_{g=h+1}^{U} \frac{C_{i_g}}{M_g^{\pi_i}}}_{\substack{\text{tail addends} \\ (b)}} \right) \tag{11}$$

Let π_{UP} be the U-long upper bound permutation for tail π_j. The upper bound that we compute for value $\Gamma(\pi_i)$ is the value resulting in calculating function Γ on input π_{UP}. Hence, using (11) we have

$$\Gamma_{UP}(\pi_j) \equiv \Gamma(\pi_{UP}) = M \cdot \left(\underbrace{\sum_{g=1}^{h} \frac{\hat{C}_{i_g}}{M_g^{\pi_{UP}}}}_{\substack{\text{head addends} \\ (a')}} + \underbrace{\sum_{g=h+1}^{U} \frac{C_{i_g}}{M_g^{\pi_{UP}}}}_{\substack{\text{tail addends} \\ (b')}} \right) \tag{12}$$

We make the following observations concerning the upper bound in (12) with respect to function Γ in (11):

- the numerators in the tail addends in (b') and in (b) are identical: they are the completion times of the tasks in the tail π_j;
- the numerators in the head addends in (a') are the completion times of the virtual tasks in π_{UP} and form an increasing sequence (with respect to the summation index). The numerators in (a') are the completion times of the tasks in the head of π_i and, in general, are not ordered;
- all denominators are M-contributions: in the upper bound these are computed by using permutation π_{UP}, while in function Γ by using permutation π_i.

Upper Bound to function Ω. Analogously to what we did for function Γ, we restate the definition of function Ω by separating the contributions of tasks in the head and the tail. Given the t-long tail permutation $\pi_j \in \Pi(\tau,t)$ and a G-long tail-constrained permutation π_i with tail π_j, we can restate the term $\omega(\pi_i)$ in the definition of $\Omega(\pi_i)$ given in equation (4) in the following way:

$$\omega(\pi_i) = \Gamma \cdot \left(\underbrace{\sum_{g=1}^{h} \frac{U_{i_g}}{M_g^{\pi_i} M_{g+1}^{\pi_i}}}_{\substack{\text{head addends} \\ (c)}} + \underbrace{\sum_{g=h+1}^{G} \frac{U_{i_g}}{M_g^{\pi_i} M_{g+1}^{\pi_i}}}_{\substack{\text{tail addends} \\ (d)}} \right) + \underbrace{\sum_{g=1}^{h} \frac{C_{i_g}}{M_g^{\pi_i}}}_{\substack{\text{head addends} \\ (a)}} + \underbrace{\sum_{g=h+1}^{G} \frac{C_{i_g}}{M_g^{\pi_i}}}_{\substack{\text{tail addends} \\ (f)}}. \quad (13)$$

Let π_{UP} be the upper bound permutation for tail π_j. The bound for $\omega(\pi_i)$ that we compute is the value of a function that is very similar to (13), on input π_{UP}. The main difference is the order of the numerator in the summation (c): in the upper bound we consider the utilizations of the virtual tasks in the head in inverse order (starting from the one with larger index), so that they form an increasing sequence. We define:

$$\omega_{UP}(\pi_j) \equiv$$
$$\Gamma \cdot \left(\underbrace{\sum_{g=1}^{h} \frac{\hat{U}_{i_{h-g+1}}}{M_g^{\pi_{UP}} M_{g+1}^{\pi_{UP}}}}_{\substack{\text{Head addends} \\ (c')}} + \underbrace{\sum_{g=h+1}^{G} \frac{U_{i_g}}{M_g^{\pi_{UP}} M_{g+1}^{\pi_{UP}}}}_{\substack{\text{Tail addends} \\ (d')}} \right) + \underbrace{\sum_{g=1}^{h} \frac{\hat{C}_{i_g}}{M_g^{\pi_{UP}}}}_{\substack{\text{Head addends} \\ (a')}} + \underbrace{\sum_{g=h+1}^{G} \frac{C_{i_g}}{M_g^{\pi_{UP}}}}_{\substack{\text{Tail addends} \\ (f')}}. \quad (14)$$

We make the following observations concerning the upper bound in (14) with respect to function ω in (13):

- head addends (a) and (a') are the same as in equations (11) and (12), respectively.
- the numerators in the tail addends in (d') (resp. (f')) and in (d) (resp. (f)) are identical. These are the utilizations (resp. completion times) of the tasks in the tail π_j;
- the numerators in the head addends in (c') are the utilizations of the virtual task in π_{UP} and form an increasing sequence (with respect to the summation index). The numerators in (c) are the utilizations of the tasks in the head of π_i and, in general, are not ordered.
- all denominators are functions of M-contributions: in the upper bound these are computed by using permutation π_{UP}, while in function ω by using permutation π_i.

Then, remembering that the $|\tau^{(j)}|$ values $\hat{U}_i$s are ordered in decreasing order, we take into account the smallest h such values, i.e., the last h, with indices ranging from $|\tau^{(j)}| - h + 1$ to $|\tau^{(j)}|$. The upper bound is given by a function that is very similar to the definition of $\Omega(\pi_i)$:

$$\Omega_{UP}(\pi_j) \equiv \frac{1}{M} \cdot \left(\underbrace{M - \sum_{v=1}^{h} \hat{U}_{|\tau^{(j)}|-v+1} - \sum_{v=h+1}^{G} U_{i_v}}_{(m)} \right) \cdot \omega_{UP}(\pi_j). \quad (15)$$

Observe that bounding at a single node of the tree, for both Γ and Ω, can be done in polynomial time, and we will always recompute the bound from scratch at each node.

Indeed, the computation of the bound at a child node can not be easily performed incrementally starting form the bound at its father node. In fact, consider the head of the of the upper bound permutation at a child node; when the corresponding virtual tasks change, the values $M_g^{\pi_{UP}}$s that appear in the denominators of (12) and (15) are clearly affected. But this means that such values have to be recomputed. In some cases such recomputation can be done starting from partial values of the $M_g^{\pi_{UP}}$s at the father note. Unfortunately, the partial values needed at a child differ from those at another child. Experimental evidence then showed that storing and accessing all the partial values needed requires more time than recomputing them again from scratch.

4.6.2 Execution Tree Pseudocode

The pseudocode for execution trees for both functions Γ and Ω is given in Figure 6. Input values are the permutation head length h and the tail permutation π_j, while the value of the best current solution S_{BC} is a global variable (so that each thread can access it and update it when needed). The function EXECUTIONTREE$S(h,\pi_j)$ is recursive: the base case occurs when the length of the head is equal to zero (test in Line 3); i.e., the search space corresponds to a specific single task permutation. In Lines 4-11, the value of the objective function is computed, compared with the best current solution, which is possibly updated (Line 10).

Recursion occurs in the **else** branch from Line 12 to Line 29. In Line 13 bounding is performed for the current search space to decide if branching is needed. In Lines 15-29 branching is performed and recursive calls (Line 28) on new children performed. Lines 20-25 are executed only by the main thread (started at the root) which assigns subtrees to parallel idle threads. GET_IDLE_THREAD() is a function that returns the id of the first thread that becomes idle (and stops the main thread meanwhile). ONTHREADS(thread$,h-1,(\tau_x,\pi_j))$ assigns subtree to thread *thread*, which then recursively calls EXECUTIONTREE$S(h-1,(\tau_x,\pi_j))$ (Line 28). Finally, lines 30-33 are executed only by the main thread: it waits for all other threads to terminate and returns the value of the best current solution.

4.6.3 Proving upper bounds

In this section we prove the correctness of the upper bounds defined in the previous section. In particular, we will prove the two following statements:

- For each U-long tail-constrained permutation π_i with tail π_j, we have

$$\Gamma(\pi_i) \leq \Gamma_{UP}(\pi_j). \tag{16}$$

- For each $G \in \{1,\ldots,U\}$, for each G-long tail-constrained permutation π_i with tail π_j, we have

$$\Omega(\pi_i) \leq \Omega_{UP}(\pi_j). \tag{17}$$

```
EXECUTIONTREE S(h, π_j)
 1  // Global variable S_BC is value of best current solution
 2  // h is head length, π_j is tail permutation
 3  if h = 0 // head is empty
 4      // Check if π_j is a better solution than best current
 5      if S(π_j) > S_BC
 6          // Get lock on S_BC
 7          GET_LOCK_ON(S_BC)
 8          // Check again if update is needed
 9          if S(π_j) > S_BC
10              S_BC := S(π_j)
11          RELEASE_LOCK_ON(S_BC)
12  else // Bound
13      if S_UP(π_j) > S_BC
14          // Branch
15          τ^(j) := τ \ π_j
16          while τ^(j) is not empty
17              Pick a task τ_x ∈ τ^(j)
18              // New head is one task shorter,
19              // new tail is old tail π_j appended to picked task τ_x
20              if |π_j| = 0 // Only thread in the root
21                  // Assign subtrees to idle threads
22                  // Wait for available thread
23                  thread = GET_IDLE_THREAD()
24                  // Assign computation to idle thread
25                  ONTHREAD S(thread, h − 1, (τ_x, π_j))
26              else
27                  // Continue with recursion
28                  EXECUTIONTREE S(h − 1, (τ_x, π_j))
29              τ^(j) := τ^(j) \ {τ_x}
30  if |π_j| = 0 // Only thread in the root
31      // Wait for all threads to terminate
32      WAITTERMINATIONOFALLTHREADS()
33      return S_BC
```

Fig. 6: Pseudo code for execution trees for both functions Γ and Ω.

Consider first equation (15). Observe that quantity (m) is not smaller than the multiplicative factor

$$M_{G+1}^{\pi_i} = M - \sum_{v=1}^{h} U_{i_v} - \sum_{v=h+1}^{G} U_{i_v}$$

that appears in the definition of $\Omega(\pi_i)$ in equation (4). In fact, the $\hat{U}_i$s in (m) are the smallest among all tasks that might appear in the head. Hence, to prove (17) we just need to prove that

$$\omega(\pi_i) \leq \omega_{UP}(\pi_j). \tag{18}$$

To prove upper bounds we compare summations that appear in the functions and in the corresponding upper bounds. In particular,

– to prove (16), referring to equations (11) and (12), we show that $(a) \leq (a')$ and that $(b) \leq (b')$;
– to prove (18) , referring to equations (13) and (14), we show that $(a) \leq (a'), (c) \leq (c'), (d) \leq (d')$ and $(f) \leq (f')$.

We start with the following lemma concerning M-contributions. It is not difficult to show that these contributions (and hence denominators) in the upper bounds are never larger than the corresponding ones in the objective functions.

Lemma 1 *For any $h, g \leq G$ and for each G-long tail constraint permutation $\pi_i \in c\Pi(\pi_j, G)$ with tail π_j, we have*

$$M_g^{\pi_i} \geq M_g^{\pi_{UP}}.$$

Proof Given g, we can write both $M_g^{\pi_i}$ and $M_g^{\pi_{UP}}$ separating the contribution given by the head and the (possibly empty) tail:

$$M_g^{\pi_i} = M \underbrace{- \sum_{v=1}^{h} U_{i_v}}_{\substack{\text{Head} \\ \text{contribution}}} \underbrace{- \sum_{v=h+1}^{g-1} U_{i_v}}_{\substack{\text{Tail} \\ \text{contribution}}} \geq M \underbrace{- \sum_{v=1}^{h} \hat{U}_{i_v}}_{\substack{\text{Head} \\ \text{contribution}}} \underbrace{- \sum_{v=h+1}^{g-1} U_{i_v}}_{\substack{\text{Tail} \\ \text{contribution}}} = M_g^{\pi_{UP}},$$

$$\sum_{v=1}^{h} U_{i_v} \leq \sum_{v=1}^{h} \hat{U}_{i_v}$$

true by definition of the $\hat{U}_i$s in the permutation π_{IP}.

This lemma allows us derive bounds for summations relative to tail contributions.

Corollary 1 *The tail contribution in equations (12) and (14) are greater or equal to tail contribution in equations (11) and (13), respectively;* i.e.*, $(b) \leq (b')$, $(d) \leq (d')$ and $(f) \leq (f')$.*

For the summations (a') and (c'), that are relative to the head contributions, we first introduce intermediate values (a'') and (c'') by substituting the M-contributions in the denominators. We have

Corollary 2 *Referring to summations in equations (12) and (14), we have*

$$(a') = \sum_{g=1}^{h} \frac{\hat{C}_{i_g}}{M_g^{\pi_{UP}}} \geq \underbrace{\sum_{g=1}^{h} \frac{\hat{C}_{i_g}}{M_g^{\pi_i}}}_{(a'')}$$

$$(c') = \sum_{g=1}^{h} \frac{\hat{U}_{i_{h-g+1}}}{M_g^{\pi_{UP}} M_{g+1}^{\pi_{UP}}} \geq \underbrace{\sum_{g=1}^{h} \frac{\hat{U}_{i_{h-g+1}}}{M_g^{\pi_i} M_{g+1}^{\pi_i}}}_{(c'')}$$

To conclude our proofs, we now just need to prove that $(a'') \geq (a)$ and $(c'') \geq (c)$. To this end, we use the following result (for the sake of presentation, its proof is postponed in Section 4.6.4).

Corollary 3 *Given two sequences of m numbers $n_1 \leq n_2 \leq \ldots n_m$ and $d_1 \geq d_2 \geq \ldots d_m$, we have*

$$\sum_{g=1}^{m} \frac{n_g}{d_g} \geq \sum_{g=1}^{m} \frac{n_{\pi(g)}}{d_g},$$

for any permutation $\pi : \{1,\ldots,m\} \rightarrow \{1,\ldots,m\}$.

Corollary 4 *The head contribution (a') in equations (12) and (14) is an upper bound to the head contribution (a) in equations (11) and (13).*

Proof We prove that

$$\sum_{g=1}^{h} \frac{\hat{C}_{i_g}}{M_g^{\pi_i}} = (a'') \geq (a) = \sum_{g=1}^{h} \frac{C_{i_g}}{M_g^{\pi_i}},$$

and the statement then follows from Corollary 2, as $(a) \geq (a'') \geq (a')$.

Assume first that the sets of numerators in (a'') and (a) coincide; *i.e.*, $\cup_{g=1}^{h}\{C_{i_g}\} = \cup_{g=1}^{h}\{\hat{C}_{i_g}\}$. Then, by Corollary 3, the thesis follows by setting $n_g = \hat{C}_{i_g}$ and $d_g = M_g^{\pi_i}$. In fact, the $\hat{C}_{i_g}$s form an increasing sequence of numbers, while the $M_g^{\pi_i}$ a decreasing one.

Assume now that there exists an index k such that C_{i_k} does not appear in the set $\cup_{g=1}^{h}\{\hat{C}_{i_g}\}$. Then, by definition of the $\hat{C}_{i_g}$s, we have $C_{i_k} \leq \hat{C}_{i_1} \leq \hat{C}_{i_k}$, and consequently that

$$\frac{\hat{C}_{i_k}}{M_k^{\pi_i}} \geq \frac{C_{i_k}}{M_k^{\pi_i}}.$$

Hence, we just need to prove that $(a'') - \frac{\hat{C}_{i_k}}{M_k^{\pi_i}} \geq (a) - \frac{C_{i_k}}{M_k^{\pi_i}}$, and we can proceed in the same way.

Lemma 2 *The head contribution (c') in equation (14) is an upper bound to the head contribution (c) in equation (13).*

Proof The proof is analogous to the proof of Corollary 4 by observing that the numerators in (c') are in increasing order.

Theorem 2 *For any G-long tail constraint permutation $\pi_i \in c\Pi(\pi_j, G)$ with tail π_j, such that $G \in \{1,\ldots,U\}$, we have*

$$\Gamma_{UP}(\pi_j) \geq \Gamma(\pi_i) \;\; \textit{and} \;\; \Omega_{UP}(\pi_j) \geq \Omega(\pi_i).$$

4.6.4 Proof of Corollary 3

In this section we give the proof of Corollary 3.

We start with the following lemma.

Lemma 3 *Given two non decreasing sequences of m numbers* $a_1 \leq a_2 \leq \ldots a_m$ *and* $b_1 \leq b_2 \leq \ldots b_m$*, we have*

$$\sum_{g=1}^{m} a_g b_g \geq \sum_{g=1}^{m} a_{\pi(g)} b_g,$$

for any permutation $\pi : \{1,\ldots,m\} \rightarrow \{1,\ldots,m\}$.

Proof Let $M = \sum_{g=1}^{m} a_g b_g$ and assume by contradiction that M is not the maximum among all possible indices permutations. Then, there must be permutation π such that $M' = \sum_{g=1}^{m} a_{\pi(g)} b_g > M$ and such that indices, in π, are not in increasing order.

Thus, there must be at least one index i such that

$$a_{\pi(i)} > a_{\pi(i)+1},$$

while we still have that $b_i \leq b_{i+1}$. Hence, we have

$$(a_{\pi(i)} - a_{\pi(i)+1})(b_{i+1} - b_i) \geq 0,$$

and thus

$$a_{\pi(i)} b_{i+1} + a_{\pi(i)+1} b_i \geq a_{\pi(i)} b_i + a_{\pi(i)+1} b_{i+1}, \tag{19}$$

where on the right side of Equation (19) we have the contribution of the two consecutive terms in M' starting at index i

We now have two cases:

(i) if the $>$ holds in Equation (19), then, by switching $a_{\pi(i)}$ with $a_{\pi(i+1)}$ we obtain a summation strictly greater M', and this is a contradiction, as M' is supposed to be the maximum value.

(ii) If the $=$ holds, then we can switch $a_{\pi(i)}$ with $a_{\pi(i+1)}$ and we obtain the same (supposedly) maximum value M' for a permutation π' in which values at indices i and $i+1$ are in increasing order. Now, either all values are in non decreasing order, and thus $M' = M$ and the thesis holds, or we can proceed as before for any other index j such that $a_{\pi'(j)} > a_{\pi'(j)+1}$.

The proof of Corollary 3 now follows simply by observing that the sequences $n_1 \leq n_2 \leq \ldots n_m$ and $\frac{1}{d_1} \leq \frac{1}{d_2} \ldots \leq \frac{1}{d_m}$ are non decreasing.

5 Experimental results

In this section we present our experimental results. We evaluate the efficiency of our implementation of *harm-BB* [13] by performing a large number of experiments. The three main contributions of this section are listed below.

- We compare the execution times of *harm-BB* and *harm-BF*, in order to establish the improvement of *harm-BB* over a simple brute force approach. In addition, we evaluate the effectiveness of *harm-BB on the inside*, by reporting the ratio between the number of task permutations pruned and the total number of permutations that had to be generated in the absence of pruning (i.e., the number of permutations generated by *harm-BF*).
- We evaluate the parallel efficiency of *harm-BB* and show the speedup it can provide on commodity multi-core CPUs.
- We compare the computation time of *harm-BB* with that of the fastest available implementations for the other polynomial algorithms for the computation of different bounds to the tardiness of global EDF schedulers.

The results reported in this section are based on the analysis of the execution times of all the mentioned algorithms over 630000 random task sets, generated according to the distributions of utilizations and periods described in the next paragraphs. These distributions are those considered in previous works on tardiness or lateness (e.g., [10,28,11]). Here we show only a few representative cases, with respect to the very large amount of results we collected, because all the experiments performed exhibit roughly the same relative performance among the algorithms. Full results and code used in the experiments can be found in [13][3].

In the rest of this section we will first give details on the experimental setting and then present our results.

5.1 Experimental settings

Systems and task sets. We generated task sets for systems with two to eight processors. We set to eight the maximum value since this appears to be the largest number of processors for which G-EDF is an effective solution to provide SRT guarantees [3]. In particular, we generated 1000 sets of implicit-deadline periodic tasks for each number of processors $M \in \{2,\ldots,8\}$ by varying task utilizations and periods in all possible ways (for a total of 630 combinations) according to the specifications given below.

Total utilization: given $M \in \{2,\ldots,8\}$, we considered task sets with total utilizations U_{sum} in the interval $[M/2,\ldots,M$, increasing in steps of $M/10$.

Task utilization: we considered both uniform and bimodal distributions so as to characterize three different "load classes".

- For *uniform distributions*:
  - *light* distribution, with task utilizations in the interval $[0.001, 0.1]$;

[3] All experiments can be repeated by applying two patches to [25] and running an *ad-hoc* script.

  - *medium* distribution, with task utilizations in $[0.01, 0.99]$;
  - *heavy* distribution, with task utilizations in $[0.5, 0.99]$.
- For *bimodal distributions*, task utilizations were chosen uniformly in either $[0.01, 0.5]$ or $[0.5, 0.99]$, with the selection of the specific range done as follows:
  - for *light* distribution, probabilities $8/9$ and $1/9$, respectively;
  - for *medium* distribution, probabilities $6/9$ and $3/9$, resp.;
  - for *heavy* distribution, probabilities $4/9$ and $5/9$, resp.

Task periods: we considered the three following *uniform* distributions, characterized by increasing (average) periods.

- *Short* distribution, in the range $[3ms, 33ms]$;
- *Moderate* distribution, in the range $[10ms, 100ms]$;
- *Long* distribution, in the range $[50ms, 250ms]$.

For the sake of presentation, in the following we will refer to a collection of 1000 task sets, generated with the same combination of the above parameters, simply as a *group*.

Algorithms and implementations. In addition to *harm-BB* and *harm-BF*, we considered the most efficient polynomial algorithms defined for computing the other three tardiness bounds for G-EDF. In the following list, we report, for each of these algorithms, the acronym that we use for the algorithm and the bound that the algorithm computes:

- *da* Bound proved by [8].
- *cva* Bound proved by [10] using the CVA analysis with the PP Reduction Rule.
- *cva2* Bound proved by [12] using the CVA analysis with the alternative optimization rule proved by [8].[4]

For each of the above algorithms, we ran its fastest version available in [25], namely its *native* (C++) version. We selected the binary-search variant for *cva* and *cva2*. *harm-BB* has been implemented in C++, using the pthread library. The code is available and can be downloaded from [13].

Execution platforms. We executed all the algorithms on an OS X and a Linux system, equipped with

- a Dual-Core (Four logical processors) 3.1 GHz Intel Core i7-5557U CPU with 1.8 GHz DDR3 DRAM,
- a Quad-Core (Eight logical processors) 2.4GHz Intel Core i7-2760QM CPU with a 1.3 GHz DDR3 DRAM,

respectively.

[4] We name *cva2* the oldest one between the last two algorithms, just to preserve the same naming as the one used for the bounds in [26].

Measures and statistics. For conciseness, given a specific algorithm and a generated task set, we will use the wording *execution time of the algorithm for the task set* to refer to the total time needed by the algorithm to compute its target tardiness bound for all the tasks in the task set. For every generated task set and every algorithm, we measured the execution time of the algorithm for that task set. We found that, for each algorithm and each group of task sets, the variation of the execution time of the algorithm was negligible across the task sets in the group. Therefore, for simplicity, hereafter we report, for each group of task sets and each algorithm, only the average execution time of the algorithm over all the tasks in the group. We call this average quantity just *execution time of the algorithm for the group of task sets*.

Graph scale. Given the high variability of the execution times of *harm-BF* and, to a lesser extent, of *harm-BB*, for the y-axis in all execution-time graphs we use a logarithmic scale.

Number of processors and number of threads. Again for brevity, we say just *number of processors* to refer to the number of processors of the virtual system on which the execution of the generated task sets is simulated, while we say just *number of threads* to refer to the number of parallel threads of execution generated by *harm-BB* (and used by *harm-BB* to compute the harmonic bound in a parallel fashion).

5.2 Results

5.2.1 Comparison between harm-BB *and* harm-BF

We compare the brute force algorithm, *harm-BF*, with the algorithm based on the Branch-and-Bound approach, *harm-BB*, by executing both algorithms on the less advantageous platform for *harm-BB*, namely the Dual-Core CPU (see next subsection for speedups). On this platform, *harm-BB* automatically sets the number of parallel execution threads to 4, *i.e.*, to the number of logical processors. Results show that *harm-BB* makes the bound feasible to compute up to eight processors, while *harm-BF* only up to six.

For example, Figure 7 shows the execution time of *harm-BB* (tagged as *harm-BB*-4, to stress the actual number of threads) and *harm-BF* for a representative group of task sets: uniform light utilizations, long periods and total utilization equal to M. This is one of the most demanding groups of task sets for exponential algorithms as *harm-BF* and *harm-BB*, because it contains the largest task sets.

As can be seen, with six processors the exponentially-growing execution time of *harm-BF* is already two orders of magnitude higher than that of *harm-BB* for 8 processors. With 7 processors *harm-BF* is already practically unfeasible, differently from *harm-BB*, whose execution time remains below 100ms even with 8 processors.

The results for the other groups of task sets exhibit a similar time/feasibility gap.

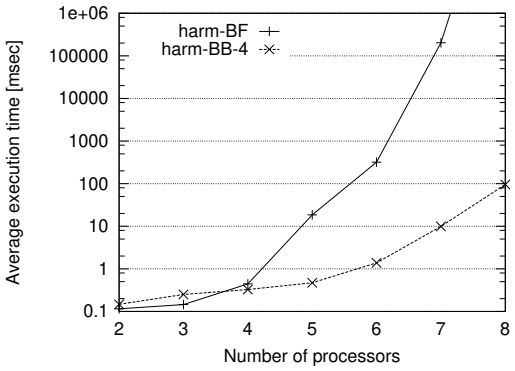


Fig. 7: Comparison between the execution times of *harm-BF* and *harm-BB* (with four threads) on the Dual-Core CPU, for the largest task-set group (uniform light utilizations, long periods and total utilization equal to M).

5.2.2 Branch-and-bound efficiency: pruning

Besides parallelism (that will be dealt with in the next section), the other reason why *harm-BB* is much faster than *harm-BF* is its effectiveness in pruning vast portions of the search space. To give evidence of this statement, we measured the ratio between the number of task permutations that *harm-BB* prunes during its execution and the total number of permutations that are evaluated by *harm-BB* (and by *harm-BB* itself when instructed to avoid pruning).

Even on the demanding group of task sets detected in the previous section (i.e., uniform light utilizations, long periods, and total utilization equal to M), *harm-BB* proves to be extremely effective in pruning the search tree. Indeed, the ratio quickly grows from $\sim$0.65 with two processors, to 0.9 with three, and remains very close to 1 for every other number of processors.

The fact that the ratio tends to increase as the number of processors grows is explained by the fact that, simultaneously working on distinct subtrees of the execution tree, very good solutions are found earlier (in one of the subtree), leading to larger pruning in all subtrees.

Despite such high pruning ratio, there are still cases in which the execution time of *harm-BB* is not negligible. This is due to the fact that, even with pruning, a very large number of permutations might have to be evaluated. Table 2 shows the figures obtained for the same demanding group of task outlined before and for M ranging from 2 to 8. We observe that such numbers grow as big as 4 millions (on the average) for $M = 8$, which explains the corresponding execution times for large values of M.

5.2.3 Parallel efficiency and speedup

To analyze the gain we have in parallelizing the Branch-and-Bound approach, we compare the execution time of *harm-BB* limited to one thread (no parallelism) and of *harm-BB* allowed a larger number of threads.

Table 2: Pruning statistics for the task set with uniform light utilizations, long periods and total utilization equal to M. Average of non pruned is rounded up to closer integer.

	$M=2$	$M=3$	$M=4$	$M=5$	$M=6$	$M=7$	$M=8$
AVERAGE RATIO	0.642	0.901	0.987	0.998	0.9998	0.99998	0.999998
AVERAGE NON PRUNED	5	76	475	3,569	36,769	386,343	4,065,575
MIN NON PRUNED	2	21	73	718	7,362	117,595	1,295,928
MAX NON PRUNED	9	158	1,207	12,125	125,662	777,396	9,807,490

We test the two versions of *harm-BB* on the two execution platform, the Dual-Core CPU first, and then the Quad-Core.

In the Dual-Core platform, speedups range from $\sim$1.4 to $\sim$1.7. To present these results, we discuss two representative cases:

(i) the first case is the one with the highest execution times, and thus one that would benefit the most from a large speedup. This happens for the task-set group with bimodal light utilizations, long periods and total utilization equal to M. This case, together with all the other cases with high execution times, happens to be also one of those with the highest speedups ($\sim$1.7). This is rather lucky, as the cases with the highest execution times are evidently the most critical ones.

(ii) The second case is the one in which *harm-BB* exhibits its lowest speedup. This happens for the task-set group with bimodal medium utilizations and long periods. The highest speedup reached by *harm-BB*, in this case, is $\sim$1.4.

Figure 8 refers to case (i) and shows the execution times of *harm-BB* in two different configurations: *harm-BB*-1, i.e., *harm-BB* limited to only one thread (no parallelism), and *harm-BB*-4, i.e., *harm-BB* allowed to generate 4 parallel threads. The latter case is the one for which *harm-BB* reaches its highest speedup on the Dual-Core platform.

Unfortunately, it is not really easy to evaluate the speedup from Figure 8, the graph results compressed because of the high variability of execution times. For this reason, in Figure 9 we report speedups explicitly, for the heaviest sub-case in Figure 8, namely 8 processors, and as a function of the number of threads that *harm-BB* is allowed to spawn. That is, for each number i of threads in the x-axis in Figure 9, we executed *harm-BB* with the number of threads (manually) set exactly to i. Figure 9 shows that *harm-BB* reaches its maximum speedup with a number of threads equal to the number of logical processors exposed by the CPU in the execution platform. For this number of threads, *harm-BB* reaches a speedup of 1.7.

For case (ii), we have the lowest speedup (namely 1.36) but we also have that the absolute execution times (given in details in Figure 13) are much lower than those for case (i). In addition, the shape of the graph of the speedups for this case is the same as in Figure 9, and hence here omitted.

We observe that in both cases the speedup is lower than the number of cores of the Dual-Core CPU platform. Hence, the speedup of *harm-BB* is evidently sub-optimal. We thus concentrate on the possible causes of this undesired behavior. In particular, we are interested in understanding if this depends on the intrinsic nature of *harm-BB*,

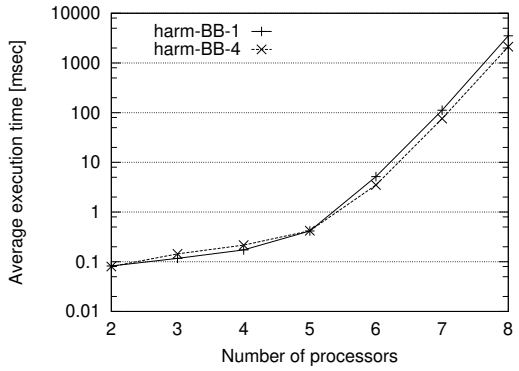


Fig. 8: Execution times of *harm-BB* on the Dual-Core platform, with 1 and 4 parallel threads, for the task-set group causing *harm-BB*'s highest execution times (bimodal light utilizations, long periods and total utilization equal to M).

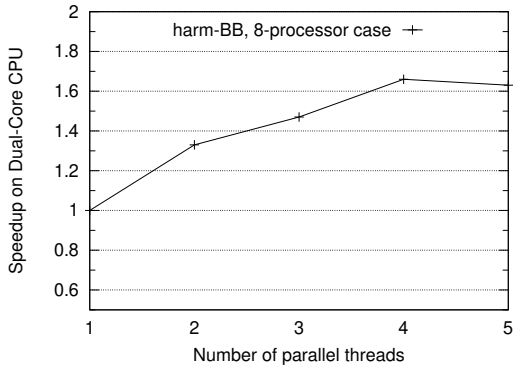


Fig. 9: Speedups of *harm-BB* on the Dual-Core platform, as a function of the number of execution threads, for the 8-processor case of the task-set group causing *harm-BB*'s highest execution times (bimodal light utilizations, long periods and total utilization equal to M).

on wrong choices made in the design of the parallel algorithm, or on the peculiarities of the execution platform. Our investigation brings us to conclude that the last one is the main reason behind sub-optimality of speedup.

The principal explanation of why the speedup is not closer to the actual number of cores of the platform is that some threads take more time than others to complete their task. To investigate if this happens during the execution of *harm-BB*, we evaluate the maximum percentage of time, over the total time needed to compute Γ or Ω (equations (3) and (5)), during which less than the total number of threads of *harm-BB* are working in parallel. In particular, we concentrate on the only two possible causes for this lack of parallelism:

Unbalanced threads. That is, threads work for different amounts of time with respect to each other. This would cause some of the threads to finish later than others, with the computational power of the CPU less and less utilized as shorter threads finish in advance. To evaluate the extent of such a load displacement in *harm-BB*, we measure the duration of each thread during the execution of *harm-BB* and verify that they all last virtually the same amount of time in our experiments.

Thread synchronization. Synchronization is needed when updating the best current solution, as in Figure 6. We have instrumented the code to count the frequency at which the check at line 5 in Figure 6 is performed, and the frequency at which the lock is actually taken. For both operations, the resulting frequency is about 7 times per second. Such a low frequency causes an absolutely negligible loss of parallelization, as each synchronization takes at most a few microseconds on the execution platforms.

This analysis hints at parallel inefficiency being somewhere else, with respect to *harm-BB*'s algorithm and implementation. Yet, should we have overlooked some aspect, and should the inefficiency that limits *harm-BB*'s speedup be actually intrinsic to *harm-BB* itself, then, by Amdahl's law, *harm-BB*'s speedup should not grow further if *harm-BB* is executed on a CPU with a higher parallelism than the Dual-Core of the first platform. As a consequence, if, on the opposite end, the speedup does grow when moving to a CPU with a higher parallelism, we have to conclude that *harm-BB*'s speedup is being limited by the execution platform.

The results in figures 10 and 11 provide an answer for this question. These figures are the counterparts, for the Quad-Core CPU, of figures 8 and 9. As shown in Figure 11, the maximum speedup is reached again with 4 threads (although the number of logical processors is twice as before). More importantly, the speedup:

1. grows linearly with the number of threads, up to when the number of threads is not higher than the number of cores;
2. reaches 3.38, *i.e.*, twice the speedup for the Dual-Core case.

As a conclusion, according to our experimental results and in-code measurements, *harm-BB* exhibits a high parallel efficiency, with a speedup limited mainly by parallel inefficiencies in the execution platform (possibly in the pthread library, operating system, CPU...).

5.2.4 Comparison among harm-BB*,* da*,* cva *and* cva2

We compare *harm-BB* against competitors and we report only results for the Dual-Core platform, as this is the platform on which *harm-BB* achieves its lowest speedup. In these tests *harm-BB* is free to generate its desired number of threads[5] and, therefore, in the following figures we label *harm-BB* with just the *harm-BB* tag.

For all groups of task sets, with the exception of the ones with light utilizations, *harm-BB* happens to be faster than, or comparable to, all the other algorithms but *da*, up to 6 processors. However, for 7 and 8 processors, *harm-BB* execution time is still

[5] according to Figure 3 and to the fact that the Dual-Core CPU exposes four logical CPUs, this means that *harm-BB* generated 4 threads.

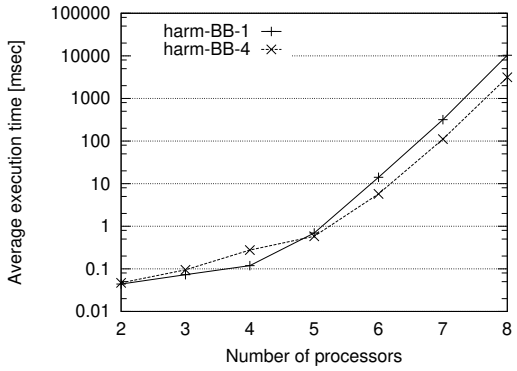


Fig. 10: Execution times of *harm-BB* on the Quad-Core platform, with 1 and 4 parallel threads, for the task-set group causing *harm-BB*'s highest execution times (bimodal light utilizations, long periods and total utilization equal to M).

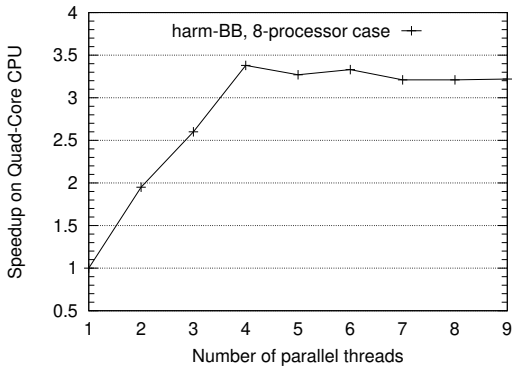


Fig. 11: Speedups of *harm-BB* on the Quad-Core platform, as a function of the number of execution threads, for the 8-processor case of the task-set group causing *harm-BB*'s highest execution times (bimodal light utilizations, long periods and total utilization equal to M).

feasible (and allows to compute a bound that is tighter than the ones computed by competitors).

Before presenting our results in more details, we observe that, for what concerns the performance of *da*, unfortunately we realized that the percentage of tasks for which *da* completes the computation of its target bound becomes lower and lower as M increases (we did not investigate this issue further). This is likely to deceptively reduce the execution time of *da* to a large extent. In contrast, *cva* and *cva2* always compute their target bounds for all tasks. Because of this, in the next comments and in the descriptions of the figures we focus only on *cva* and *cva2*.

Figure 12 shows the execution times of the four algorithms in the case of long periods of uniform heavy utilization and total utilization equal to M. This is the best

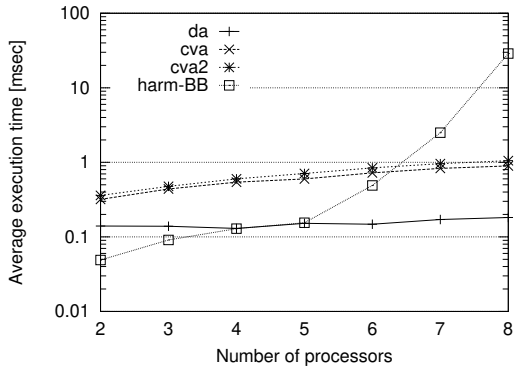


Fig. 12: Comparison among the execution times of *harm-BB*, *da*, *cva* and *cva2* for one of the groups of task sets for which *harm-BB* exhibits its lowest execution times (uniform heavy utilizations, long periods and total utilization equal to M).

case for *harm-BB*, even if its performance still reflects *harm-BB* general behavior, worsening significantly above 6 processors. However, for this task-set group the execution times of *harm-BB* remains in the order of the tens of milliseconds, hence still feasible. The same order of magnitude holds for all task-set groups with not too many tasks and with not too much variable utilization.

Figure 13 shows the execution times of the four algorithms for a task-set group with a moderate number of tasks and a moderate variability of utilizations. As can be seen, the execution times of *harm-BB* with 7 and 8 processors become much higher than those of the competitors and of the previous case. This group represents an intermediate-performance example for *harm-BB*.

The worst groups, in terms of utilization variability and number of tasks, are those with bimodal light distributions. Figure 14 shows the group, in this subset, for which *harm-BB* exhibits its worst performance. This is the same group for which we have already shown speedups in the previous section (Figure 8). Also for these task sets, *harm-BB* is significantly slower than the other algorithms only for 7 and 8 processors, but it is still feasible.

According to our experiments, we conclude that the groups of task sets for which *harm-BB* exhibits its highest execution times are those with both a high variability of utilizations and a high number of tasks.

Indeed, the number of tasks influences the cardinality of the search space for *harm-BB*, and therefore its execution time, as the size of the search space is factorial in the number of tasks. However, the variability of utilizations is the most important factor affecting the execution time of *harm-BB*. The explanation to this phenomenon is to be found in the fact that such variability of utilizations rapidly decreases the effectiveness of *harm-BB* pruning strategy.

This happens when the upper bound computed for a given subset of feasible solutions is too loose: the subset is not pruned because its upper bound is larger than the

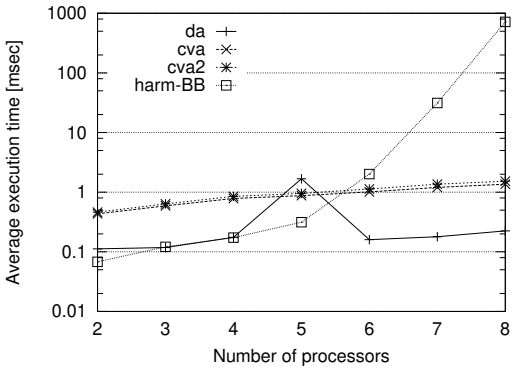


Fig. 13: Comparison among the execution times of *harm-BB*, *da*, *cva* and *cva2* for one of the groups of task sets for which *harm-BB* exhibits an intermediate performance (bimodal medium utilizations, long periods and total utilization equal to M).

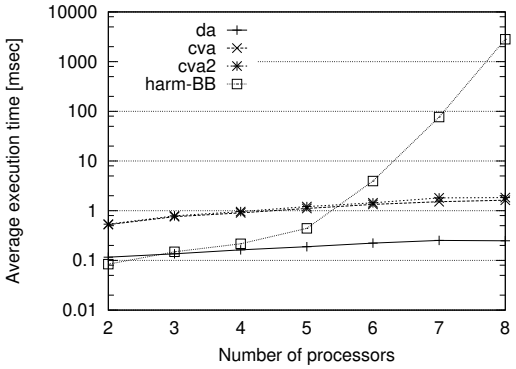


Fig. 14: Comparison among the execution times of *harm-BB*, *da*, *cva* and *cva2* for the largest-size group of task sets for which *harm-BB* exhibits its worst performance (bimodal light utilizations, long periods and total utilization equal to M).

value of the best current solution, even if there is no solution in the subset that leads to a value larger than the best current solution.

In *harm-BB*, this undesired behavior might occur because the choice of the $\hat{U}$s in the summation $\hat{m} = \left(\sum_{v=1}^{h} \hat{U}_{|\tau^{(j)}|-v+1}\right)$ of the term (m) in (15) is independent from choice of the utilizations $\hat{U}$s in the head contribution (c') in the term $\omega_{UP}(\pi_j)$ in (14). When utilizations presents a large variability, the utilizations chosen for $\hat{m}$ might be small (leading to a large (m)), while those chosen for $\omega_{UP}(\pi_j)$ might be large (leading to a large $\omega_{UP}(\pi_j)$). Therefore, the value of $\Omega_{UP}(\pi_j)$ (resulting from the product of the former terms) gets greatly inflated.

6 Conclusions

We run experiments on 630000 task sets and, for almost all of them, our implementation of *harm-BB* proved able to compute the harmonic bound for the tardiness of G-EDF schedulers in less, or almost the same, time than what takes existing polynomial algorithms to compute looser tardiness bounds. Only for the highest numbers of processors (7 and 8) and tasks ($\sim$50), *harm-BB* was slower than the other algorithms. It was however still feasible, with an execution time not higher than 3 seconds in the worst cases. We also detected the cases in which *harm-BB* performs poorly with respect to competitors, *i.e.*, instances with a large number of tasks and large variability of task utilizations. Such information might be used when deciding which algorithm to use to compute the bound (always keeping in mind that *harm-BB* computes a tighter bound).

Moreover, we showed that *harm-BB* has a high parallel efficiency, which allows for a large cut down of its execution time on a highly-parallel platform (as, for example, the same one deemed to execute the applications for which the bound is computed), making *harm-BB* always a valid alternative to competitors.

Acknowledgments

Authors wish to thank Prof. Daniele Funaro for providing a compact proof of Lemma 3.

References

1. Anderson, J.H., Srinivasan, A.: Mixed pfair/erfair scheduling of asynchronous periodic tasks. Journal of Computer and System Sciences **68**(1), 157 – 204 (2004). DOI 10.1016/j.jcss.2003.08.002. URL http://www.sciencedirect.com/science/article/pii/S0022000003001508
2. Baruah, S.K., Cohen, N.K., Plaxton, C.G., Varvel, D.A.: Proportionate progress: A notion of fairness in resource allocation. Algorithmica **15**, 600–625 (1996)
3. Bastoni, A., Brandenburg, B., Anderson, J.: An empirical comparison of global, partitioned, and clustered multiprocessor real-time schedulers. In: Proceedings of the 31st IEEE Real-Time Systems Symposium, pp. 14–24 (2010)
4. Bastoni, A., Brandenburg, B.B., Anderson, J.H.: An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In: Proceedings of the 2010 31st IEEE Real-Time Systems Symposium, RTSS '10, pp. 14–24. IEEE Computer Society, Washington, DC, USA (2010). DOI 10.1109/RTSS.2010.23. URL http://dx.doi.org/10.1109/RTSS.2010.23
5. Brandenburg, B.B., Gül, M.: Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In: 2016 IEEE Real-Time Systems Symposium (RTSS), pp. 99–110 (2016). DOI 10.1109/RTSS.2016.019
6. Cavicchioli, R., Capodieci, N., Bertogna, M.: Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms. In: 22nd IEEE International Conference on Emerging Technologies And Factory Automation (ETFA) (2017)
7. Devi, U.C., Anderson, J.H.: Tardiness bounds under global edf scheduling on a multiprocessor. In: RTSS, pp. 330–341. IEEE Computer Society (2005). URL http://dblp.uni-trier.de/db/conf/rtss/rtss2005.html#DeviA05
8. Devi, U.C., Anderson, J.H.: Tardiness bounds under global edf scheduling on a multiprocessor. Real-Time Systems **38**(2), 133–189 (2008)
9. Eckstein, J., Phillips, C.A., Hart, W.E.: Pico: An object-oriented framework for parallel branch and bound. In: D. Butnariu, Y. Censor, S. Reich (eds.) Inherently Parallel Algorithms in Feasibility and Optimization and their Applications, *Studies in Computational Mathematics*, vol. 8, pp.

219 – 265. Elsevier (2001). DOI https://doi.org/10.1016/S1570-579X(01)80014-8. URL http://www.sciencedirect.com/science/article/pii/S1570579X01800148
10. Erickson, J.P., Anderson, J.H.: Fair lateness scheduling: Reducing maximum lateness in g-edf-like scheduling. In: ECRTS, pp. 3–12 (2012)
11. Erickson, J.P., Anderson, J.H., Ward, B.C.: Fair lateness scheduling: reducing maximum lateness in g-edf-like scheduling. Real-Time Systems **50**(1), 5–47 (2014). DOI 10.1007/s11241-013-9190-4
12. Erickson, J.P., Devi, U., Baruah, S.K.: Improved tardiness bounds for global edf. In: ECRTS, pp. 14–23 (2010)
13. Experiment-scripts: Code used for experiments. (2014). URL http://algogroup.unimore.it/people/paolo/harmonic-bound/
14. Horowitz, E., Sahni, S.: Fundamentals of Computer Algorithms. Computer Science Press (1978)
15. Kenna, C.J., Herman, J.L., Brandenburg, B.B., Mills, A.F., Anderson, J.H.: Soft real-time on multiprocessors: Are analysis-based schedulers really worth it? In: RTSS, pp. 93–103. IEEE Computer Society (2011). URL http://dblp.uni-trier.de/db/conf/rtss/rtss2011.html#KennaHBMA11
16. Kumar, V., Kanal, L.N.: Parallel branch-and-bound formulations for and/or tree search. IEEE Transactions on Pattern Analysis and Machine Intelligence **PAMI-6**(6), 768–778 (1984). DOI 10.1109/TPAMI.1984.4767600
17. Lai, T.H., Sprague, A.: Performance of parallel branch-and-bound algorithms. IEEE Transactions on Computers **C-34**(10), 962–964 (1985). DOI 10.1109/TC.1985.6312201
18. Leoncini, M., Montangero, M., Valente, P.: A branch-and-bound algorithm to compute a tighter bound to tardiness for preemptive global edf scheduler. In: Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS '17, pp. 128–137. ACM, New York, NY, USA (2017). DOI 10.1145/3139258.3139277. URL http://doi.acm.org/10.1145/3139258.3139277
19. Li, J., Agrawal, K., Lu, C., Gill, C.: Analysis of global edf for parallel tasks. In: 2013 25th Euromicro Conference on Real-Time Systems, pp. 3–13 (2013). DOI 10.1109/ECRTS.2013.12
20. Liu, C., Anderson, J.H.: Supporting soft real-time parallel applications on multiprocessors. Journal of Systems Architecture **60**(2), 152 – 164 (2014). DOI http://dx.doi.org/10.1016/j.sysarc.2013.07.001. URL http://www.sciencedirect.com/science/article/pii/S1383762113001227
21. Maia, C., Yomsi, P.M., Nogueira, L., Pinho, L.M.: Semi-partitioned scheduling of fork-join tasks using work-stealing. In: 2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing, pp. 25–34 (2015). DOI 10.1109/EUC.2015.30
22. Megel, T., Sirdey, R., David, V.: Minimizing task preemptions and migrations in multiprocessor optimal real-time schedules. 2013 IEEE 34th Real-Time Systems Symposium **0**, 37–46 (2010). DOI http://doi.ieeecomputersociety.org/10.1109/RTSS.2010.22
23. Mills, A., Anderson, J.: A stochastic framework for multiprocessor soft real-time scheduling. In: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE, pp. 311–320 (2010). DOI 10.1109/RTAS.2010.33
24. Regnier, P., Lima, G., Massa, E., Levin, G., Brandt, S.A.: Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In: RTSS, pp. 104–115 (2011)
25. SchedCAT: The schedulability test collection and toolkit (2014). URL https://github.com/brandenburg/schedcat/
26. Valente, P.: Using a lag-balance property to tighten tardiness bounds for global edf. Real-Time Systems **52**(4), 486–561 (2016). DOI 10.1007/s11241-015-9237-9. URL http://dx.doi.org/10.1007/s11241-015-9237-9
27. Valente, P., Lipari, G.: An upper bound to the lateness of soft real-time tasks scheduled by edf on multiprocessors. IEEE 26th Real-Time Systems Symposium **0**, 311–320 (2005). DOI http://doi.ieeecomputersociety.org/10.1109/RTSS.2005.8
28. Ward, B.C., Erickson, J.P., Anderson, J.H.: A linear model for setting priority points in soft real-time systems. In: Proceedings of Real-Time Systems: The Past, the Present, and the Future, pp. 192–205 (2013)
29. Yang, K., Anderson, J.H.: Optimal gedf-based schedulers that allow intra-task parallelism on heterogeneous multiprocessors. In: 2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia), pp. 30–39 (2014). DOI 10.1109/ESTIMedia.2014.6962343
30. Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., Sha, L.: Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In: 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 55–64 (2013). DOI 10.1109/RTAS.2013.6531079