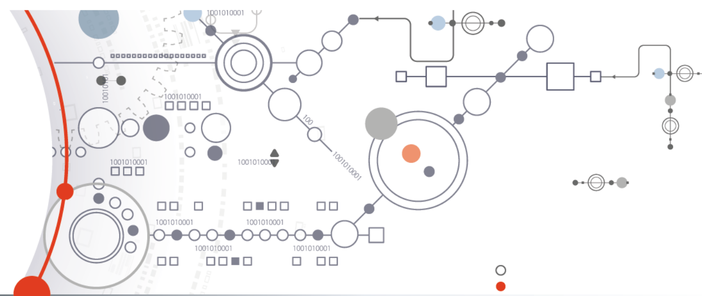

Scaling Up the Memory Interference Analysis for Hard Real-Time Many-Core Systems (Full Version)

Maximilien Dupont de Dinechin[*], *Matheus Schuh*[†‡], *Matthieu Moy*[§], *Claire Maiza*[¶]

Verimag Research Report n^o TR 2019-1

November 2019

[*] *Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP* F-69342, LYON Cedex 07, France first.last@univ-lyon1.fr
[†] *Univ. Grenoble Alpes CNRS, Grenoble INP, VERIMAG* 38000 Grenoble, France first.last@univ-grenoble-alpes.fr
[‡] *Kalray* Montbonnot-Saint-Martin, France first.last@kalray.eu
[§] *Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP* F-69342, LYON Cedex 07, France first.last@univ-lyon1.fr
[¶] *Univ. Grenoble Alpes CNRS, Grenoble INP, VERIMAG* 38000 Grenoble, France first.last@univ-grenoble-alpes.fr

Reports are downloadable at the following address
http://www-verimag.imag.fr

Unité Mixte de Recherche 5104 CNRS - Grenoble INP - UGA

Bâtiment IMAG
Université Grenoble Alpes
700, avenue centrale
38401 Saint Martin d'Hères
France
tel : +33 4 57 42 22 42
fax : +33 4 57 42 22 22
http://www-verimag.imag.fr/

Scaling Up the Memory Interference Analysis for Hard Real-Time Many-Core Systems (Full Version)

Maximilien Dupont de Dinechin[*], *Matheus Schuh*[†‡], *Matthieu Moy*[§], *Claire Maiza*[¶]

November 2019

Abstract

The emergence of many-core architectures has raised interest even from the embedded hard real-time market for their performance and integrity capabilities. However, to use such processors in these safety-critical systems, the software running on it must have statically bounded execution time. Due to the presence of multiples cores, interference may appear when accessing shared resources, increasing the overall run-time and diminishing predictability.

In RTNS 2016, Rihani et al. [7] proposed an algorithm to compute the impact of interference on memory accesses on the timing of a task graph. It calculates a static, time-triggered schedule, i.e. a release date and a worst-case response time for each task. The task graph is a DAG, typically obtained by compilation of a high-level dataflow language, and the tool assumes a previously determined mapping and execution order. The algorithm is precise, but suffers from a high $\mathcal{O}(n^4)$ complexity, n being the number of input tasks. Since we target many-core platforms with tens or hundreds of cores, applications likely to exploit the parallelism of these platforms are too large to be handled by this algorithm in reasonable time.

This paper proposes a new algorithm that solves the same problem. Instead of performing global fixed-point iterations on the task graph, we compute the static schedule incrementally, reducing the complexity to $\mathcal{O}(n^2)$. Experimental results show a reduction from 535 seconds to 0.90 seconds on a benchmark with 384 tasks, i.e. 593 times faster.

Keywords: response time analysis, algorithm optimization, many-core architectures, real-time systems

Reviewers: Lionel Reig

Notes: This is a long version of an accepted paper at DATE 2020, that extends it with more details about the architecture used as an example and some ideas for for formally proving the algorithm. The first author is also affiliated to ENS Paris - PSL, reachable at the following e-mail address: maximilien.dupont.de.dinechin@ens.fr

[*] *Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP* F-69342, LYON Cedex 07, France first.last@univ-lyon1.fr

[†] *Univ. Grenoble Alpes CNRS, Grenoble INP, VERIMAG* 38000 Grenoble, France first.last@univ-grenoble-alpes.fr

[‡] *Kalray* Montbonnot-Saint-Martin, France first.last@kalray.eu

[§] *Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP* F-69342, LYON Cedex 07, France first.last@univ-lyon1.fr

[¶] *Univ. Grenoble Alpes CNRS, Grenoble INP, VERIMAG* 38000 Grenoble, France first.last@univ-grenoble-alpes.fr

How to cite this report:

@techreport {TR 2019-1,
title = {Scaling Up the Memory Interference Analysis for Hard Real-Time Many-Core Systems (Full Version)},
author = { Maximilien Dupont de Dinechin, Matheus Schuh, Matthieu Moy, Claire Maiza },
institution = {{Verimag} Research Report},
number = {TR 2019-1},
year = {2019}
}

1 Introduction

Programs running in safety-critical real-time embedded systems must remain predictable in terms of execution time to meet the engineering constraints in their specification. Avionics or autonomous vehicles applications, for example, have analysis and decision making in code heavily coupled to time, so each task in the system must be temporally tightly bounded. Usually, these programs are made of periodic loops that activate tasks and any timing deviation might be propagated causing overlapping issues and even functional failure.

For many reasons (energy, performance, integrity, availability), embedded systems are shifting from single-core to multi/many-cores platforms. A many-core is a type of architecture that typically has hundreds of cores and whose computational power mainly relies on the parallelism level of the programs it runs, in contrast with multi-core processors, where a unique core can be quite powerful on its own. In this work we use the Kalray MPPA-256 [3] as the evaluation many-core platform, specifically its second generation, called Bostan. It has 256 cores distributed evenly into 16 clusters and useful architectural mechanisms for real-time systems, such as banked memory and simple arbitration. More details about the processor are given in Section 2.2.

The use of these new architectures raises challenges in the way general purpose programming is conceived, and even more for the real-time domain. Particularly, we are interested in computing a program's global Worst-Case Execution Time (WCET) and analyzing how multiple cores may impact this value. In single-core processors, there is only one entity accessing the memory at a given time, and the WCET in isolation is sufficient to estimate the global one through simple addition. With multiple cores, two tasks running simultaneously in distinct cores cannot be granted access to the memory at the same time, and therefore they slow each other down. Such a slowdown is called interference.

In [4] a framework to develop time-predictable real-time systems for many-core architectures is introduced. It is composed of multiple stages, starting with a dataflow application, which is divided into smaller computational blocks that are compiled into `C` code, resulting in a DAG of tasks, partially ordered by their dependencies. For each task, the WCET in isolation and number of memory accesses are obtained through a tool such as OTAWA [2]. Subsequently the tasks are mapped to cores and ordered. In the final step, the release dates and Worst-Case Response Time (WCRT), i.e. WCETs taking interference into account, are computed.

Contribution This paper presents a new algorithm to compute this last step of the framework in $\mathcal{O}(n^2)$ time for a program divided into n subtasks. Its implementation is done in `Python` using the Kalray MPPA-256 as target platform, but conceived with generalization in mind, so new architectures can be integrated. The improvement from previous works [6] and [7] is huge, where an algorithm to solve this problem was showcased, but with a $\mathcal{O}(n^4)$ time complexity making it intractable for very large task graphs.

Organization Section 2 provides a more in-depth description of the briefly introduced problem, elaborating on the expected input, output and hypotheses assumed. In Section 3 the original solution from [6] is explored, leveraging its potential optimization points that are used in Section 4 where our solution is detailed. Section 5 contains some hints about the proof of termination and correctness of the new algorithm. To conclude, in Section 6 a complexity analysis and a performance evaluation of the implementation are given.

2 Context

This section provides an explanation of the interference effect, a description of the processor studied and finally defines the concurrency problem introduced by many-core architectures that must be solved to properly bound the execution time of software running on it.

2.1 Interference due to arbitration

Hardware arbiters handle how accesses to a shared resource from different initiators are ordered. Multiple types of arbitration policies exist, serving different purposes, such as timing predictability or throughput. The shift to many-core architectures makes the memory bus arbiter a major influence on the execution time of programs.

A simple, deterministic and starvation-free arbitration policy is the Round-Robin (RR), found in the memory bus arbiter of the MPPA-256. It gives each initiator an equal grant share in circular order, conditioned by the use of this share. This means that cores access the memory one after another, as long as all of them are requesting to read or write data, otherwise they are skipped.

For instance, assuming a bus size of width 1 word with RR arbitration policy, if three cores have to write 8 words to the memory, the first one writes 1 word, then the second one 1 word, then the third one 1 word and this process is repeated until no core needs to write anymore. In a concrete scenario, the first core to get its access granted suffers no interference, but a very detailed analysis would be needed to know which core is delayed and which one is not. Instead, we consider the worst case in the analysis, i.e. that all cores are delayed. With this policy, all three cores are halted 8+8 times, and assuming that each word access takes 1 cycle, they each receive a total worst-case interference of 16 cycles.

2.2 Kalray MPPA-256

The Kalray MPPA-256 is a many-core processor, composed of 16 Compute Clusters (CCs). Each of these clusters has 16 Processing Engines (PEs) and an additional Resource Manager (RM). The cores are composed of an in-order Very Long Instruction Word (VLIW) pipeline, that allows instruction level parallelism while maintaining a predictable execution time. Each core also has its private data and instruction caches. The connection with the external environment is done through its 4 I/O subsystems. The inter-cluster communication is possible via a 2D-torus dual Network-On-Chip (NoC). Our application model is restricted to the CC, only considering the interference among tasks running on the same cluster accessing the shared memory [5].

The clusters have a local memory of 2MBytes, accessible by all cores (or PEs). To provide spatial isolation and minimize interference, this memory is partitioned into 16 independently arbitrated banks. This means that cores that access different memory banks go through different arbiters, hence do not interfere with each other. In this study, we consider a fixed association between cores and memory banks so that tasks running on the same core access the same memory bank. As a consequence, `read` requests are private but `write` requests can go to another's core memory bank and cause interference.

2.3 General description of the problem

To precisely estimate the interference, we need to know the time interval during which memory accesses will be performed by each core. For this, we use a time-triggered schedule, where tasks, running on cores, are assigned a release date rel (i.e. the task cannot start before this date even if all its inputs are available earlier), and a WCRT R is computed. As a consequence, we can guarantee the absence of interference between two tasks when their execution interval $[rel,\ rel + R]$ has no overlap.

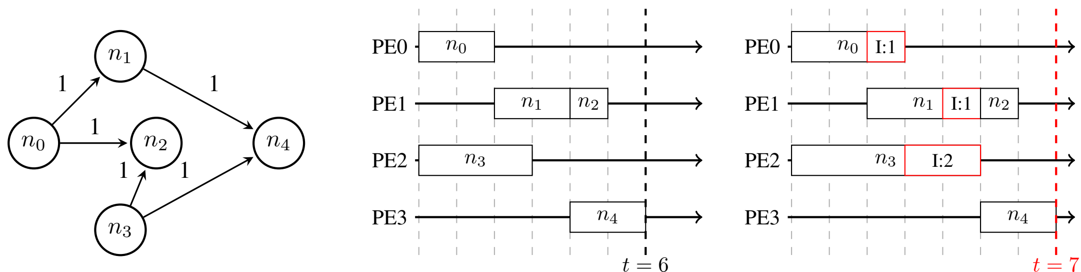

Figure 1: Minimalist program mapped to 4 cores and its timing schedules

Given a Directed Acyclic Graph (DAG) of tasks with dependencies, their mapping and schedule onto cores, their WCETs in isolation, their memory accesses and the bus arbiter description, we need to compute release dates for each of those tasks and the total WCRT of the graph, which accounts for the interference between tasks simultaneously accessing (either reading from or writing to) the shared memory. Additionally, some tasks may have a minimal release date, meaning that they must not be scheduled before that date.

The difficulty in solving this problem is that the release dates and interference values are dependent on each other. This means that modifying the release dates of tasks can change how they interfere with each other and a new amount of interference might change the release date of yet to be scheduled tasks. However, once a solution is found, the computed release dates allow to always maintain a precise execution: even if the dependencies of a task are executed faster than their WCETs, the task will not be released before, avoiding unexpected interferences.

Figure 1 shows an example of a task set, its initial schedule on the middle, and its final schedule accounting for interference, on the right. The mapping is the following: $n_0 \mapsto$ PE0; $n_1, n_2 \mapsto$ PE1; $n_3 \mapsto$ PE2 and $n_4 \mapsto$ PE3. Their WCETs in isolation are respectively 2, 2, 1, 3 and 2. Moreover, there are minimal release dates defined: $t = 0$ for n_0, n_3; $t = 2$ for n_1 and $t = 4$ for n_2, n_4. The amount of memory write accesses can be seen in the DAG on the edges between the nodes. In the timing diagram we can see the interference impact on the release dates and WCRT of the tasks, resulting in a global WCRT of $t = 7$, instead of $t = 6$ when the interferences are ignored.

In the next section we discuss some non-trivial assumptions that allow us to later develop the algorithm using the basic concepts of the problems described here.

2.4 Approximations and hypotheses

We assume the following relatively weak constraint: adding a new task to the program can only increase the interference received by other tasks. This is an intuitive statement, yet necessary for the proof of correctness of the proposed algorithm later on. For generality purposes, we assume that the interference might be *non additive*, meaning that the interference between n tasks is not necessarily the sum of the interferences between all pairs of tasks. However, some bus arbiters have this *additivity* property, and exploiting this could simplify and speed up the algorithm for those cases.

We then add a conservative hypothesis: when multiple tasks are mapped to the same core, they can be treated as a single big task, summing their WCETs, and memory accesses. This hypothesis empirically outputs less pessimistic release times than a more complex approach consisting in computing all the disjoint sets of tasks interfering with a given one. Figure 2 shows two diagrams that are topologically the same when considering a task set where n_1 and n_2 interfere with n_0. They are perceived as only one big task by task n_0.

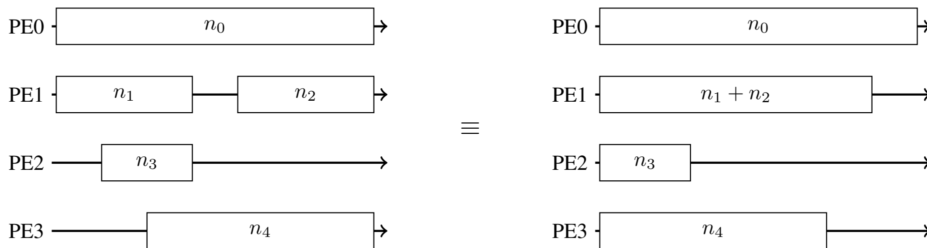

Figure 2: Equivalence between diagrams for interference calculation. Tasks n_1 and n_2 can be grouped from the point of view of task n_0

3 Original algorithm

We cover here in more detail how the algorithm upon which we base our work is constructed as well as which improvements points were investigated in our version.

Algorithm 1: Original scheduling algorithm

Input: Set of release dates $\Theta = \{rel_1, \ldots, rel_n\}$, set of response times $\mathcal{R} = \{R_1, \ldots, R_n\}$ of tasks $\{\tau_1, \ldots, \tau_n\}$
Output: `schedulable`, Θ, $\mathcal{R}$ **OR** `unschedulable`

1 $l \leftarrow 0$, $\Theta^l \leftarrow$ InitRel(), $\mathcal{R}^l \leftarrow \perp$;
2 **do**
3 $\quad \mathcal{R}^{l+1} \leftarrow$ ComputeResponseTimes(Θ^l);
4 $\quad \Theta^{l+1} \leftarrow$ UpdateReleaseDates($\Theta_{min}, \Theta^l, \mathcal{R}^{l+1}$);
5 $\quad l \leftarrow l + 1$;
6 **while** $\Theta^l \neq \Theta^{l-1}$;

3.1 Description

In [1], an algorithm is proposed to compute a bound on the delay due to interference for a set of sporadic tasks. It served as an inspiration for the algorithm introduced in [7], which we improve in this paper. [7] uses two fixed-point iterations to compute the global response-time. The first iteration computes the interference between all tasks with a given set of release dates. The second one adjusts all release dates to respect the dependencies. They are repeated until a stable value for the release dates is found or the deadline is crossed, meaning that the task set is unschedulable. A simplified version is shown in Algorithm 1.

This algorithm was proved to have a $\mathcal{O}(n^4)$ complexity [6] where n is the number of tasks to schedule, which raises scalability issues. The goal of this work is to reduce this complexity allowing it to be applied to hundreds of tasks.

3.2 Improvement approach

The functions in line 3 and 4 of Algorithm 1 have a complexity of $\mathcal{O}(n^3)$ (n iterations to converge, n iterations to calculate the response time of all tasks and $n-1$ iterations to compute the interference function) and $\mathcal{O}(ne)$ (iterates over all tasks and its dependencies, which are the graph edges), respectively, and therefore are extremely costly. The new algorithm takes some ideas from the termination proof in [6]: a time cursor is used to prove that at each iteration a new task gets its definitive release date. Then, instead of computing fixed-points, our algorithm simply iterates through the finite task set and the dependencies with this time cursor.

4 Proposed algorithm

The new algorithm is introduced in this section, starting with its main idea, then exploring its inner working details with a complete pseudo code.

4.1 Core idea

Given the task set and initial release dates, the proposed algorithm works incrementally, by adding tasks one by one to the schedule. The algorithm works with a time cursor t, starting from $t = 0$ and progressing forward. The tasks are divided into three groups:

- Closed ($\mathcal{C}$): t is after their finish date. These tasks have both their final release date and response times computed.
- Alive ($\mathcal{A}$): t is between their release and finish date. These tasks have their final release date, but their response time may be influenced by tasks not yet added to the schedule.
- Future: t is before their release date, neither the release date nor their response time is computed.

At each iteration, the cursor t jumps to the nearest end date of the current alive tasks or the minimal release of future tasks, whichever is smaller. New available tasks, i.e. with all dependencies satisfied, are

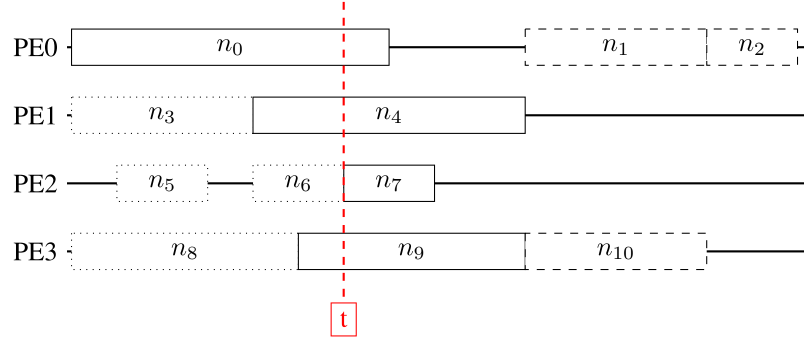

Figure 3: Snapshot of the new algorithm cursor mechanism

then scheduled, and the interferences that they add to and receive from the current alive tasks are computed. They cannot interfere with dead tasks, because they do not overlap, and their interferences with future tasks will be computed later in the algorithm, when those are added to the alive group.

With this approach, when a task is scheduled, its release date is definitively set and, as previously discussed, will not be changed with future tasks.

Figure 3 captures a snapshot of the algorithm being executed. The vertical dashed red line represents the current time cursor position. Only the solid boxes are considered alive tasks. The dotted boxes on the left are the dead tasks, and the dashed ones on the right are the future tasks.

4.2 Detailed algorithm

The proposed algorithm is given in Algorithm 2 as pseudo code, and detailed below. The inputs are a task set Γ, an initial set of release dates Θ and response times $\mathcal{R}$, the number of cores c available in the platform, the mapping of tasks to cores and a shared memory, which may have distinct arbitrated banks reserved for each core to minimize interference[6].

In the example from Figure 3, we have $\Gamma = \{n_0, \ldots, n_{10}\}$, $c = 4$ and the mapping is as follows: $n_0, n_1, n_2 \mapsto$ PE0, $n_3, n_4 \mapsto$ PE1, $n_5, n_6, n_7 \mapsto$ PE2 and $n_8, n_9, n_{10} \mapsto$ PE3.

The time cursor begins at $t = 0$, with $\mathcal{A}$, the set of current alive tasks, initially empty. The following steps are then repeated until all the tasks are scheduled (at each step we give the corresponding state in the example from Figure 3 and the lines from the Algorithm 2):

1. $\mathcal{C}$ (closed) is the set of tasks ending at time t. It is simply computed by scanning the current alive tasks, and determining if the end of the task (rel + WCRT) equals t. These tasks are then removed from their reverse dependencies list, allowing tasks depending on these closed ones to start.

   Algorithm: Lines 3 to 6, *Example:* $\mathcal{C} = n_6$

2. $\mathcal{A}$ (Alive) $\leftarrow \mathcal{A} - \mathcal{C}$

   Algorithm: Line 7, *Example:* $\mathcal{A} = n_0, n_4, n_9$

3. $\mathcal{O}$ (Opening) is the set of tasks opening at time t. It is computed by scanning the head of the stack of scheduled tasks for each core, and determining whether its dependencies are satisfied and if its minimal release date is smaller than or equal to t.

   Algorithm: Lines 9 to 15, *Example:* $\mathcal{O} = n_7$

4. $\mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{O}$

   Algorithm: Line 16, *Example:* $\mathcal{A} = n_0, n_4, n_7, n_9$

[6] If the memory is composed of only one bank (or the banks are interleaved and memory accesses cannot reliably be associated to a bank), the loop iterating over banks on lines 22 to 26 of Algorithm 2 iterates over only one element and can be simplified. However, while the analysis is simplified, the worst-case interference is increased as all access from all cores, even to their private code and data, may be delayed by others.

Algorithm 2: Proposed scheduling algorithm

Input: Set of release dates $\Theta = \{rel_1, \ldots, rel_n\}$, set of response times $\mathcal{R} = \{R_1, \ldots, R_n\}$ of tasks $\{\tau_1, \ldots, \tau_n\}$
Output: `schedulable`, Θ, $\mathcal{R}$ **OR** `unschedulable`

```
1  forall k, S_k ← stack of tasks scheduled on core k; A ← ∅; t ← 0;
2  while t < +∞ do
3      C ← {τ ∈ A | (τ.rel + τ.WCET + τ.inter) = t};
4      for τ ∈ C do
           // τ.rev_deps → tasks that depend on τ
5          for τ' in τ.rev_deps do
6              τ'.deps.remove(τ);
7      A ← A − C;
8      O ← ∅;
9      for k ∈ list of cores c = {0, ..., c − 1} do
10         if S_k is not empty then
               // get top of stack without removing
11             τ_next ← S_k.peek();
12             if τ_next.deps is empty AND
                min_rel of τ is ≤ t then
13                 O ← O ∪ {τ};
14                 τ.rel ← t;
15                 S_k.pop();                                   // removes top of stack
16     A ← A ∪ O;
17     for τ_dest ∈ A do                                        // task target of mem access
18         for τ_src ∈ A do                                     // task source of access
19             for bank b in banks B do
20                 if τ_dest and τ_src both access b then
21                     if τ_src not in τ_dest.interfers_with[b] then
22                         τ_dest.interfers_with[b].add(τ_src);
23                         τ_dest.interferences[b] ← I^BUS(τ_dest, τ_dest.interfers_with[b], b);
24     t_next ← +∞;
25     for τ ∈ A do
26         t_next ← min(t_next, τ.rel + τ.WCET + τ.inter);
27     for min_rel in minimal release of future tasks do
28         t_next ← min(t_next, min_rel);
29     t ← t_next;
```

5. For any *destination* task in $\mathcal{A}$ and for any *source* task in $\mathcal{A}$, which have accesses to the same memory bank, we determine if the source task has already been accounted for in the interferences received by the destination. If not, that interference is recomputed by the specified bus arbiter function (see Section 6.2), after adding the source of the list of nodes that the destination interferes with. Notice that at least one of the source and destination task must be in $\mathcal{O}$, otherwise it would already have been accounted for. The interferences are computed separately for each memory bank access from the task τ. The total interference received by the task τ is the sum of those values.

   Algorithm: Lines 17 to 23

6. t is updated to the minimal value between the next smallest release date of future tasks and the next finish time of alive tasks.

   Algorithm: Lines 25 to 29

5 Termination and correctness

Termination Except the outermost on t all the loops are iterating over finite sets, which by definition ensures that they will end. The main while loop finishes too, as there are at most $2n$ possible values for t_next: each jump of t is to the beginning or end of a task, and those, once visited, will never change later.

Correctness Assume the algorithm is correct when scheduling the $n-1$ first tasks. When the n-th task is added, its release date is definitively set and it may still only interferes the alive tasks. Once the interference computation is done, the algorithm finishes and, by recurrence, all previous dependencies and release dates are respected, providing a correct and final schedule for the task set.

Equivalence While this algorithm is conjectured to be equivalent to the original one, according to the result of the tests performed, this remains to be formally proven.

6 Results

6.1 Complexity

The size of the set of alive tasks $\mathcal{A}$ is bounded by the number of cores. Therefore, we access the linear I^{BUS} function (line 23 of Algorithm 2 and explained in the next section) a bounded number of times for each progression of t, and the possible values for t are tasks end dates and their minimal release dates, making it at most $2n$. The two nested loops then give an overall complexity, with n tasks, b banks and c cores:

$$O(c^2 \cdot b \cdot n^2) \tag{1}$$

For a given processor, b and c are constants, so we may simplify Equation (1) to $\mathcal{O}\big(n^2\big)$.

6.2 Bus arbiter function

The algorithm discussed in Section 4 is conceived to be modular with regard to the interference model. For this reason, the bus arbiter function $I^{\text{BUS}}(\tau, \mathcal{S}, b)$ is completely decoupled from the main algorithm. It takes a task τ, a set $\mathcal{S}$ of tasks it interferes with and the bank b where those interferences take place. The output is the interference received by the task τ, i.e. the number of cycles that the task is delayed.

This function abstracts the hardware arbiters found in processors. As the MPPA-256 was used for our study its RR arbitration function is described here and used during the performance evaluation.

The RR arbitration was introduced in Section 2 as a relatively simple and predictable policy. It gives a fair quota to each initiator. Denote by $a_b(\tau)$ the number of accesses (read or write) of task τ on bank b. The interference received by a task τ is then at most:

$$I^{\text{BUS}}(\tau, \mathcal{S}, b) = \sum_{\tau' \in \mathcal{S}} \min\left(a_b(\tau'), a_b(\tau)\right) \tag{2}$$

The left term of the min comes from the fact that τ cannot be halted more than the number of another task's accesses, and the right term because τ cannot be slowed down more than the number of times it accesses the memory.

We might add refinements to the modeling of RR by grouping tasks that are running on the same core in the min function, as from our previous assumptions in Section 2.4, they act as one long task from the perspective of τ. With n cores, and Λ the function that maps tasks to their core:

$$I^{\text{BUS}}(\tau, \mathcal{S}, b) = \sum_{i=1}^{n} \min\left(\sum_{\substack{\tau \in \mathcal{S} \\ \Lambda(\tau')=i}} a_b(\tau'), a_b(\tau) \right) \tag{3}$$

6.3 Experimental results

To compare the old and new algorithm on real world scenarios, we generate random DAGs, using a method proposed by Tobita and Kasahara in [9], explained and used in the original work by Rihani [6]. Figure 4 gives an overview of the methodology.

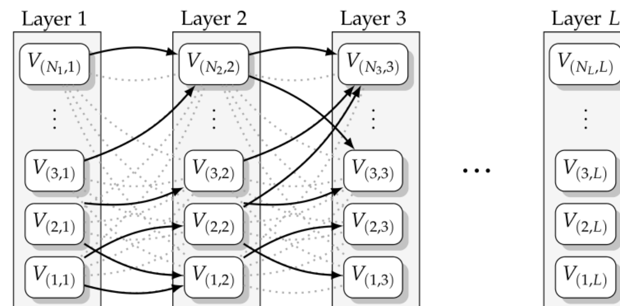

Figure 4: Random DAG generation, method from [9] and image from [6].

This method is called *layer-by-layer* DAG generation. Tasks on the same layer are assigned to the cores in a cyclic way: the n-th task of a layer is assigned to the core $c = (n \mod \textit{number of cores})$. Tasks have randomly generated WCET, memory accesses and write operations on tasks of the next layer, respectively between the values [550, 650], [250, 550] and [0, 100]. Two approaches are used to generate the inputs of the benchmark: fixed NL, in which the number of layers is constant and the layer size increases, and fixed LS, in which the layer size stays the same and it is the number of layers that gets enlarged.

The implementation of the original algorithm is done in `C++`, while the proposed algorithm is written in `Python`. This means that there is an interpreter overhead that negatively impact our results mainly for a small number of tasks.

	LS4	LS16	LS64	NL4	NL16	NL64
`Python` (new)	1.03	1.02	1.10	1.75	1.89	1.91
`C++` (original)	3.71	4.39	5.09	4.52	4.64	4.94

Table 1: n^x complexity comparison

A linear regression computation on a $\log \times \log$ scale from the benchmark values was done to see if the theoretical complexity goes in hand with the practical outcome. Table 1 shows the results, where NL4 represents a fixed number of layers of 4, and LS4 a fixed layer size of 4. The bus arbiter function used is the Kalray MPPA-256 RR from [6], depicted in Equation (3) with a maximum allocation of the tasks to 16 cores. All the benchmark parameters are the same as in [6].

The complexity of the proposed algorithm always stay under $\mathcal{O}(n^2)$, contrary to Rihani's which exceeds $\mathcal{O}(n^4)$ and even seems to reach $\mathcal{O}(n^5)$ in the NL64 and LS64 cases.

These results are plotted in Figure 5, revealing the drastic improvement in computation time provided by the `Python` version. The benchmark has a timeout that the `C++` version easily reaches for more than 256 tasks.

In particular, as seen in Table 1, LS64 and NL64 are the random DAGs configuration values that showcase the biggest difference between the two versions. For LS64 and 256 tasks, the `C++` version took $1121.79s$ and the `Python` one took mere $4.13s$, or 270 times faster. For N64 and 384 tasks, the `C++` implementation executed for $535.24s$ and the `Python` for only $0.9s$, or 593 times faster.

7 Conclusion

This paper introduces a new algorithm to obtain the release dates and response times of applications in the context of real-time systems implemented on many-core architectures. The revisited version shows

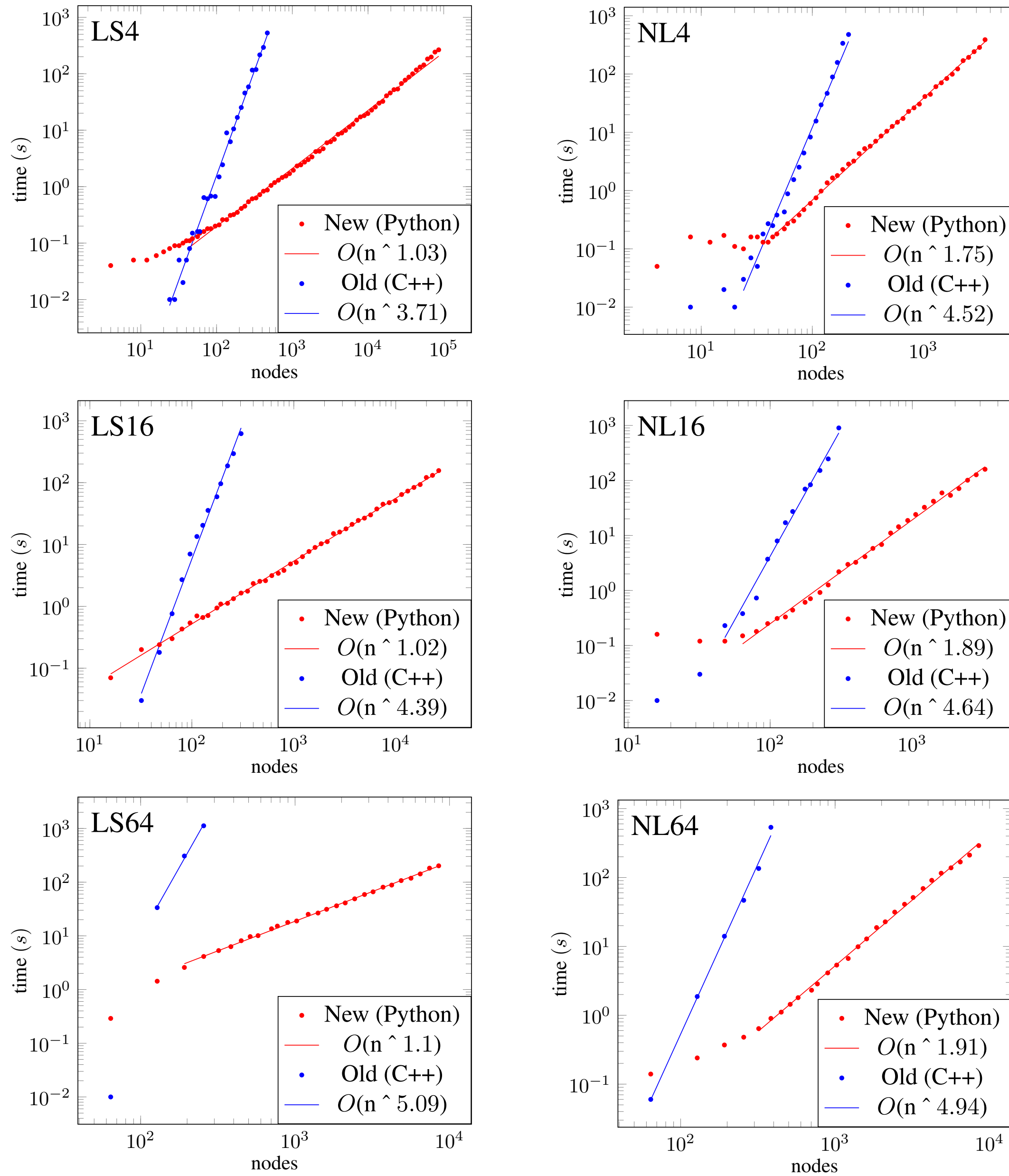

Figure 5: Benchmark plotted results

a significative complexity improvement to $\mathcal{O}(n^2)$, which translates to 593 times faster runtime in our benchmark, in comparison with the original version from [7]. This allows to accomplish the requirements of modern safety-critical real-time systems, scaling to more than 8000 tasks while maintaining a reasonable execution time.

Future work include overlap management to provide a finer upper bound on the interference and thus less pessimistic results. It is based upon the idea that tasks will only be able to access the memory at the same time and interfere during their overlap period. Another work path is to develop a method to dynamically schedule the tasks accounting for interferences. This combines the scheduling, ordering and response time analyzer steps in the framework from [4]. This idea was given by Rouxel et al. in [8] where they state that the lower complexity of the response time analyzer algorithm might allow it to be merged with the scheduler to implement predictable and optimized programs on many-core architectures.

Glossary

CC Compute Cluster

DAG Directed Acyclic Graph

I/O Input/Output

NoC Network-On-Chip

PE Processing Engine

RM Resource Manager

RR Round-Robin

VLIW Very Long Instruction Word

WCET Worst-Case Execution Time

WCRT Worst-Case Response Time

References

[1] Sebastian Altmeyer, Robert I Davis, Leandro Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. A generic and compositional framework for multicore response time analysis. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pages 129–138. ACM, 2015. 3.1

[2] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: an open toolbox for adaptive wcet analysis. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2010. 1

[3] Benoît Dupont De Dinechin, Duco Van Amstel, Marc Poulhiès, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014. 1

[4] Amaury Graillat. *Code Generation for Multi-Core Processor with Hard Real-Time Constraints*. Theses, Université Grenoble Alpes, November 2018. 1, 7

[5] Vincent Nélis, Patrick Meumeu Yomsi, and Luís Miguel Pinho. The variability of application execution times on a multi-core platform. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016. 2.2

[6] Hamza Rihani. *Many-Core Timing Analysis of Real-Time Systems*. Theses, Université Grenoble Alpes, December 2017. 1, 1, 6, 3.2, 6.3, 4, 6.3

[7] Hamza Rihani, Matthieu Moy, Claire Maiza, Robert I Davis, and Sebastian Altmeyer. Response time analysis of synchronous data flow programs on a many-core processor. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 67–76. ACM, 2016. (document), 1, 3.1, 7

[8] Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. Tightening contention delays while scheduling parallel applications on multi-core architectures. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):164, 2017. 7

[9] Takao Tobita and Hironori Kasahara. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling*, 5(5):379–394, 2002. 6.3, 4