Scaling Up the Memory Interference Analysis for Hard Real-Time Many-Core Systems

Maximilien Dupont de Dinechin[1, *]
[1]*Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP*
F-69342, LYON Cedex 07, France
first.last@univ-lyon1.fr

Matheus Schuh[2,3]
[2]*Univ. Grenoble Alpes*
CNRS, Grenoble INP, VERIMAG
38000 Grenoble, France
first.last@univ-grenoble-alpes.fr

Matthieu Moy[1]
Claire Maiza[2]
[3]*Kalray*
Montbonnot-Saint-Martin, France
first.last@kalray.eu

***Abstract*—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.

***Index Terms*—response time analysis, algorithm optimization, many-core architectures, real-time systems**

I. 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, but the algorithm can deal with other arbitration policies.

We are interested in computing a program's global Worst-Case Execution Time (WCET) and analyzing how multiple cores may impact this duration: 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 [5] 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. A long version of this paper with more details on the algorithms and results is available in [4].

Organization: Section II presents the problem, elaborating on the expected input, output and hypotheses assumed. In Section III the original solution from [6] is briefly explored before the Section IV where our solution is detailed. To conclude, in Section V a complexity analysis and a performance evaluation of the implementation are made.

II. Context

A. 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.

* This author is also affiliated to ENS Paris - PSL, reachable at the following e-mail address: maximilien.dupont.de.dinechin@ens.fr

978-3-9819263-4-7/DATE20/©2020 EDAA

A simple, deterministic and starvation-free arbitration policy is the Round-Robin (RR). It gives each initiator an equal grant share in circular order, conditioned to 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 interference of 16 cycles.

B. General description of the problem

To precisely estimate the interference, we need to know the time interval during which memory accesses are 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.

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 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 (top) and then its final schedule accounting for interference (bottom). 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

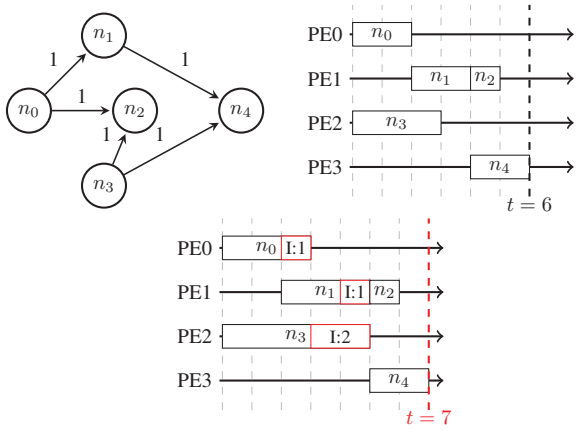

Figure 1. Minimalist program mapped to 4 cores and its timing schedules

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 ar 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.

C. Approximations and hypotheses

We assume the following constraints: adding a new task to the program can only increase the interference received by other tasks; and for generality purposes, the interference might be *non additive*, meaning that the interference between n tasks is not necessarily the sum of the interferences between all pairs. However, some bus arbiters have this *additivity* property, and exploiting this could simplify and speed up the algorithm for those cases.

Also, we 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.

III. Original algorithm

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.

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.

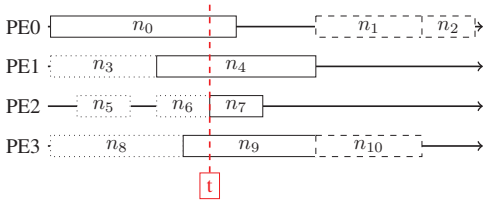

Figure 2. Snapshot of the new algorithm cursor mechanism

IV. Proposed algorithm

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 date of future tasks, whichever is smaller. New available tasks, i.e. with all dependencies satisfied, are 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.

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. The key idea behind the complexity reduction is that only tasks in $\mathcal{A}$ need to be considered in the interference analysis.

Figure 2 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 ones on the right are the future tasks.

A. Detailed algorithm

The proposed algorithm is given in Algorithm 1 as pseudo code, and detailed below. The inputs are a task set Γ, their initial release dates Θ and response times $\mathcal{R}$, the number of cores c available in the platform, how the tasks are mapped onto them and a shared memory, that may have distinct arbitrated banks reserved for each core to minimize interference.

In the example from Figure 2, 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 2):

Algorithm 1: 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, $S_k \leftarrow$ stack of tasks scheduled on core k;
  $\mathcal{A} \leftarrow \varnothing$; $t \leftarrow 0$;
2 **while** $t < +\infty$ **do**
3 $\mathcal{C} \leftarrow \{\tau \in \mathcal{A} \mid (\tau.\text{rel} + \tau.\text{WCET} + \tau.\text{inter}) = t\}$;
4 **for** $\tau \in \mathcal{C}$ **do**
    // τ.rev_deps → tasks that depend on τ
5 **for** τ' *in* τ.rev_deps **do**
6 τ'.deps.`remove` (τ);
7 $\mathcal{A} \leftarrow \mathcal{A} - \mathcal{C}$;
8 $\mathcal{O} \leftarrow \varnothing$;
9 **for** $k \in$ *list of cores* $c = \{0, \ldots, c-1\}$ **do**
10 **if** S_k *is not empty* **then**
      // get top of stack without removing
11 $\tau\_next \leftarrow S_k$.`peek`();
12 **if** τ_next.deps *is empty* ***AND*** min_rel *of* τ *is* $\leq t$ **then**
13 $\mathcal{O} \leftarrow \mathcal{O} \cup \{\tau\}$;
14 τ.rel $\leftarrow t$;
15 S_k.`pop`(); // removes top of stack
16 $\mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{O}$;
17 **for** $\tau\_dest \in \mathcal{A}$ **do** // task target of mem access
18 **for** $\tau\_src \in \mathcal{A}$ **do** // task source of access
19 **for** *bank* b *in banks* $\mathcal{B}$ **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] $\leftarrow$ $I^{\text{BUS}}(\tau\_dest, \tau\_dest.\text{interfers\_with}[b], b)$;
24 $t\_next \leftarrow +\infty$;
25 **for** $\tau \in \mathcal{A}$ **do**
26 $t\_next \leftarrow \min(t\_next, \tau.\text{rel} + \tau.\text{WCET} + \tau.\text{inter})$;
27 **for** min_rel *in minimal release of future tasks* **do**
28 $t\_next \leftarrow \min(t\_next, min\_rel)$;
29 $t \leftarrow t\_next$;

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. *Example:* $\mathcal{C} = n_6$
2) $\mathcal{A}$ (Alive) $\leftarrow \mathcal{A} - \mathcal{C}$ *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. *Example:* $\mathcal{O} = n_7$
4) $\mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{O}$ *Example:* $\mathcal{A} = n_0, n_4, n_7, n_9$
5) For any *destination* and *source* task in $\mathcal{A}$ that access 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 bus arbiter function, after adding the *source* to the list of nodes that the *destination* interferes with. 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.

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.

B. 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 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: $\mathcal{O}\left(c^2 \cdot b \cdot n^2\right)$. For a given processor, b and c are constants, so we may simplify this equation to $\mathcal{O}\left(n^2\right)$.

V. 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 [8], explained and used in the original work by Rihani [6].

This method is called *layer-by-layer* DAG generation. Tasks on the same layer are assigned to cores in a cyclic way: the n-th task of a layer is assigned to Core($n \mod$ *number of cores*). Tasks have randomly generated WCET, memory accesses and write operations on tasks of the next layer, respectively between [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 mean that there is an interpreter overhead that negatively impact our results mainly for a small number of tasks.

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. Figure 3 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]. The complexity of the proposed algorithm always stay under $\mathcal{O}\left(n^2\right)$, contrary to Rihani's which exceeds $\mathcal{O}\left(n^4\right)$ and even seem to reach $\mathcal{O}\left(n^5\right)$ in the NL64 and LS64 cases. The benchmark has a timeout that the `C++` version easily reaches for more than 256 tasks.

In particular, 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.

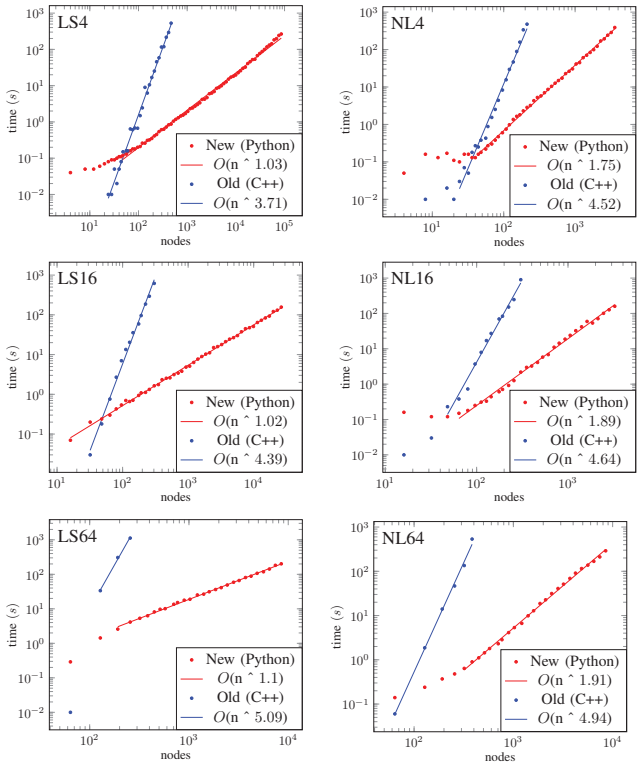

Figure 3. Benchmark plotted results

VI. 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 a significative complexity improvement to $\mathcal{O}\left(n^2\right)$, 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.

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 *RTNS*, pages 129–138, 2015.

[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.

[3] Benoît Dupont De Dinechin, Duco Van Amstel, Marc Poulhiès, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In *DATE*, pages 1–6. IEEE, 2014.

[4] Maximilien Dupont de Dinechin, Matheus Schuh, Matthieu Moy, and Claire Maiza. Scaling up the memory interference analysis for hard real-time many-core systems (full version). Technical report, Verimag Research Report TR-2019-1, 2019.

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

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

[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 *RTNS*, pages 67–76. ACM, 2016.

[8] 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.