Published in IEEE Design & Test, 2018, vol. 35, no 4, pp. 8-15, which should be cited to refer to this work. DOI : 10.1109/MDAT.2017.2783326

Safe and Efficient Deployment of Data-Parallelisable Applications on Many-Core Platforms: Theory and Practice

Stefanos Skalistis[1], Federico Angiolini[1], Alena Simalatsar[2], and Giovanni De Micheli[1]

[1]*Integrated Systems Laboratory (LSI), École Polytechnique Fédérale de Lausanne (EPFL)*
[2]*Institute of Systems Engineering, HES-SO Valais-Wallis, Email: {firstname.lastname}@{epfl|hevs}.ch*

***Abstract*—The *safe* and, at the same time, *efficient* deployment of parallelisable applications on many-core platforms is a challenging task. Theoretical Models of Computation (MoC) require the realistic estimation of task Worst-Case Execution Time (WCET) to provide safe latency guarantees. Due to interferences on shared resources, task WCET estimations are often exceedingly pessimistic. In reality, though, rarely do all the tasks execute with their WCET, thus introducing an efficiency gap, which is of consequence in realizing safety-critical and mixed-criticality systems. In this paper, we outline the additional research efforts required to i) derive a safe deployment from a MoC reducing that efficiency gap and ii) adapt at runtime to further improve performance and still preserve safety. We also outline the impact of the level of data-parallelisation onto this efficiency gap and present experimental evidence of the performance improvements from accurate WCET estimation, level of data-parallelisation and runtime adaptation.**

I. Introduction

Data-parallelisable applications [1], often represented with *data-flow* Models of Computation (MoC) [2], [3], are suitable for many-core architectures due to their high degree of task and data parallelism. These MoCs can provide useful guarantees (e.g. deadlock freedom, absence of data races), required for safety-critical systems. Nevertheless, utilizing them for hard real-time systems, which require completion deadlines, in an *efficient* manner is challenging. Engineering hard real-time systems based on theoretical MoCs is founded on the a-priori knowledge of task Worst-Case Execution Time (WCET). However, in multi-core architectures, the WCET *varies* according to the deployment, as tasks interfere when simultaneously accessing shared resources, i.e. memories, buses and processors. This effect is particularly apparent in data-parallelisable applications, due to extensive resource sharing, thus constituting a severe engineering challenge. Indeed, recent research [4], [5], [6] shows that WCET estimations which account for interferences from parallel tasks can be 150% to even 750% of the corresponding estimations in absence of interference. As an illustrative example (see Figure 1) of the WCET variability, consider four equivalent tasks, e.g. an image filter applied to four different parts of an image. These tasks, being equivalent, have the same Worst Case Computation Time (WCCT) when executed in isolation. If the overhead per each overlapping task is of one unit, it is clear that the WCET of each task will increase and additionally will vary depending on the schedule.

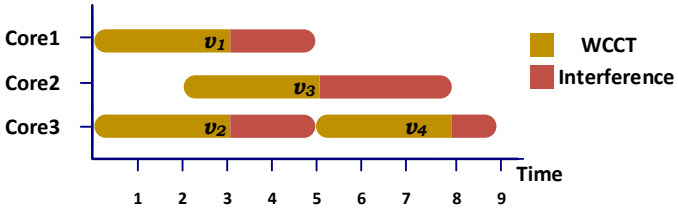


Fig. 1: Example of varying WCET for four equivalent tasks executed on three cores.

Applying MoCs, such as Synchronous Data-Flow graphs (SDFs) [2] and Kahn Process Networks (KPNs) [3], to efficiently implement actual systems is, therefore, far from straightforward. In this paper, we present the necessary research and engineering efforts to utilize such models without undermining performance. In particular we utilize our previous works on data-parallelisable applications modelled as a subclass of SDFs (but without being strictly limited to that subclass) deployed on *homogeneous NoC-based MPSoCs*. Data-parallelisable applications are particularly interesting since, if properly deployed, they can provide spatial isolation thus reducing the worst-case estimation for the interference. Having such an application, we first generate a safe deployment solution using an interference-based WCET estimation method. This method accounts for the problem of inter-task interference and provides a tighter latency guarantee, than would otherwise be impossible to acquire. Yet, even such statically-tightened WCETs are over-approximations and conservative schedules based on them result in sub-optimal system performance. This calls for adaptation at runtime based on the Actual Execution Time (AET), provided that safety is still guaranteed.

Both accurate WCET estimation and runtime adaptation are beneficial for safety-critical and mixed-criticality systems. Accurate WCET estimation results in systems with lower latency/higher throughput guarantees and improved power estimations. Runtime adaptation can yield additional improvements to power consumption, enabling e.g. longer operation when battery-powered, and performance, creating timing slack which can be exploited, in the case of mixed-criticality sys-

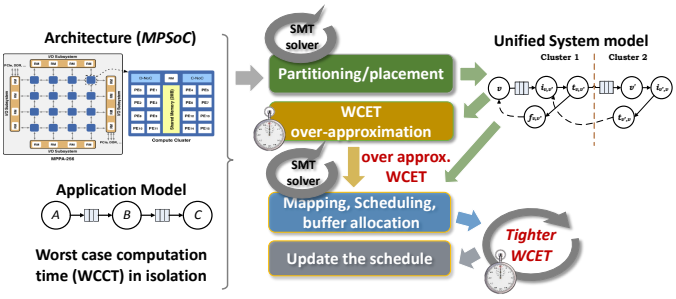


Fig. 2: Overview of our approach [5] to acquire a safe deployment.

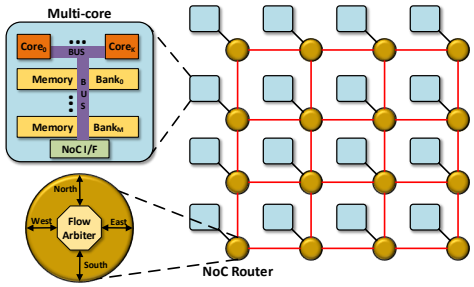


Fig. 3: Illustration of the reference many-core architecture.

tems, by best-effort tasks.

The main contributions of this paper are i) outlining research directions necessary to realize safe *and* efficient systems by presenting an exemplar software deployment flow for many-core systems, which guarantees that hard deadlines will be met, while optimising the actual execution time; ii) an analysis of the additional optimisation possible when accounting for the degree of application data parallelism; and iii) a generic low-overhead runtime adaptation technique which preserves safety.

II. Acquiring a Safe Deployment

To acquire a safe deployment for a data-parallelisable application, we utilize our previous theoretical work on safe deployment of data-flow applications on many-core architectures [4] and an interference-based WCET analysis [5]. Both methods assume that the task WCET is composed of the WCCT when executed in isolation (including baseline delays to load/store data) and the delays due to *interferences*. This distinction, which is rarely encountered in formal models, is essential in achieving efficient deployment solutions and runtime adaptation, as shown in Sections III and IV.

Using an architecure and an application model as inputs, we acquire a safe deployment solution by performing application i) *partitioning*, by balancing the computation load using the WCCT, and ii) *placement*, by assigning different partitions to compute clusters while minimising the communication delay, which depends on the distance between the communicating clusters (see Figure 2). Subsequently, we derive the *unified system model*, which encompasses both the computation and communication elements of the platform. Based on the worst-case interference, we then compute pessimistic WCETs for tasks of the unified system model, and use them to derive near-optimal solutions for the application *mapping, scheduling,* and *buffer allocation*, thus providing real-time guarantees. The solutions to the aforementioned optimisation problems are provided by an SMT solver, via sets of constraints describing the unified system model and limitations of the target platform, but other methods, e.g. any scheduling technique listed in [7], [8], [9] or state-of-the art methods for KPNs [3] or SDFs [2], can be used as well.

These solutions are then analysed using our WCET tightening methods [5]. Since a schedule is now available, it is possible to exclude inter-task interference that cannot occur because tasks that appeared to be possibly interfering are now known to not overlap either in time or resources. Applying this tightening at the end of the task mapping flow breaks the cyclic inter-dependence between deployment and interference, providing a set of efficient deployments which preserve real-time guarantees.

In the following subsections we briefly present the formal models required to derive safe deployment solutions, while referring the reader to [4], [5] for a complete discussion.

A. Platform Architecture Model

We consider a generalised many-core architecture as a tuple $P = (X, K, M, N)$ where X is the set of identical *clusters* interconnected with a NoC and K, M, N are the sets of processing *cores*, *memory banks* and *NoC channels* per cluster, respectively. Each cluster has one NoC interface, connected to a dedicated NoC router (Figure 3).

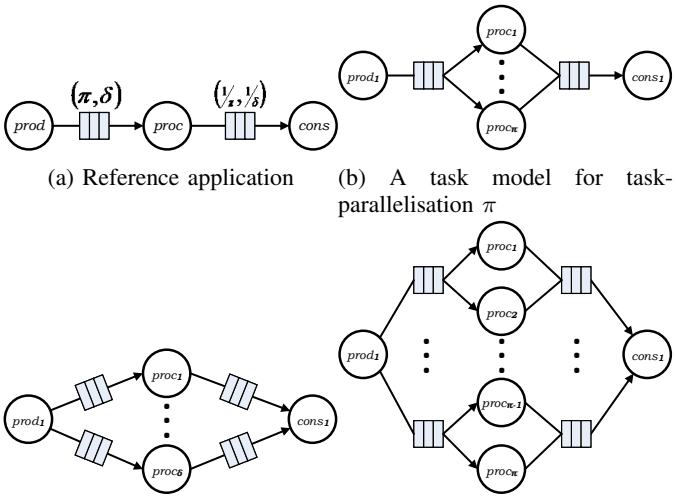
Intra-cluster data exchanges occur over the cluster *shared memory*, while inter-cluster transfers are handled by the NoC. This model is inspired by various NoC-based multi-cores, e.g. the ones considered in [10], [6], and is generic enough to model several platforms. For simplicity, we focus only on data transfers within the chip and not between the chip and main memory, Ethernet interfaces, I/O devices etc. We also assume that the platform provides a non-polling notification mechanism (e.g. interrupts) among cores of the same cluster, with bounded execution time and memory accesses.

B. Application and Unified System Models

Data-parallelisable applications are frequently modeled as SDFs [2], which provide useful properties, such as deadlock freedom, absence of data races, confluency, buffer protection, etc. In this model, *actors* exchange data via First-In-First-Out (FIFO) channels and are iteratively *fired*. At every *firing*, corresponding to a *task* execution, the actor consumes/produces data tokens from/to the input/output FIFOs, respectively.

We focus on a particular class of SDFs[1], called split-join graphs [11], which model explicitly the parallelisation of each actor. From a split-join graph, equivalent graphs can be devised depending on the data- (δ) and task- (π) parallelisation factors, as shown in Figure 4. We will always choose, from these equivalent graphs, the ones where FIFOs are split up to the

[1]The focus is due to the explicit modelling of parallelisation factors and existence of tools (StreamExplorer) for these models.



(a) Reference application (b) A task model for task-parallelisation π

(c) A task model for data-parallelisation δ (d) A task model with parallelisation $\pi > \delta$

Fig. 4: Various task models for the reference *produce-process-consume* application with *process* having task-/data-parallelisation factors of π and δ, respectively.

maximum δ, i.e the ones corresponding to Figures 4c, 4d. The reason is that splitting the FIFOs to the maximum possible degree can provide spatial isolation, thus has a significant impact on the WCET estimation as explained in Section III, and by extension on the efficiency of the solution.

An application with data dependencies is modelled as an SDF directed acyclic graph[2] $\mathcal{G} = (\mathcal{U}, \mathcal{E})$, with $\mathcal{U}$ being a set of *computation actors* and $\mathcal{E}$ a set of dependencies among them. Actors communicate via bounded FIFOs, one for each edge $e \in \mathcal{E}$ of the graph. An *application model* is an annotated SDF graph $\mathcal{A} = (\mathcal{U}, \mathcal{E}, d, \sigma)$, where the *delay function* d describes the WCCT of each firing of actor u in absence of interferences, while the *size function* σ captures the traffic volume (in words) of each edge e.

Since NoC transfer times vary according to the distance of the communicating clusters, to faithfully assess execution times on a target NoC-based platform, the *placement* function $p : U \rightarrow X$ has to be solved [5], [9]. Only then it is possible to model all data transfers with *communication tasks*, which encapsulate the arbitration, transfer and memory release stages of said communication. This leads to the *unified system model*, modelled as an annotated SDF graph $\mathcal{S} = (\mathcal{U}', \mathcal{E}', d', \sigma')$ [4], which describes the execution of the application A on a platform P accounting for both computation and communication.

C. Generic Interference Model and Safe Deployments

We model data transfers in a very general way as a sequence of word-by-word *requests*. The parallel firing of actors, henceforth *tasks*, introduces mutual interferences when requests share resources. We assume *intra*-cluster interference, i.e. where requests suffer arbitration delays, to occur at two points: i) at an intra-cluster bus, ii) at an intra-cluster memory bank. *Inter*-cluster transfers on the other hand can also run into interferences when iii) a remote transfer task collides with a memory access of an intra-cluster task, or iv) two transfer tasks compete for the same NoC router.

We assume, as in previous work [5], that the WCET of transfer tasks is bounded by guaranteed services offered by the underlying platform, like in the Kalray MPPA-256 chip [10], but this assumption can be lifted if the routing of NoC flows is known. Given a unified system model $\mathcal{S} = (\mathcal{U}, \mathcal{E}, d, \sigma)$ using our methods [4], [5] and existing tools[3] we acquire a safe deployment $(\mu_E, \mu_M, s, d_{wc})$, where the *task mapping* function (μ_E) maps tasks to platform cores or NoC channels, the *memory mapping* function (μ_M) associates graph FIFOs to physical memory banks, the *scheduling* function (s) associates a start time to each task, and the *worst-case delay* function (d_{wc}) provides a deployment-dependent WCET. We then transform the unified system model $\mathcal{S}$, as in [11], to an annotated task graph $TM = (V, E, d_{wc}, \sigma)$ called *task model*. In this transformation, V is the set of tasks and E are the data-dependencies between the tasks augmented with additional scheduling dependencies such that the FIFOs remain protected.

This is a safe deployment, in the sense that no task violates any deadlines and the total latency is guaranteed, since the WCET was originally over-approximated and then safely tightened, while maintaining buffer protection (see proofs in [4], [5]).

III. Data Parallelisation and Impact on Interference

As described above, a safe deployment for an application A can be identified. It is now important to outline the impact of inter-task interference on such a deployment, particularly in accordance with the data-parallelisation factor δ. If δ is known for every task of application A, it is possible to acquire more efficient, and still safe, solutions by estimating the interference on different task models for different values of δ.

To do so, we first report the WCET estimations for a reference application and compare them with estimations which disregard δ. In Section IV we will evaluate such effects on the actual execution of the reference application.

As a reference application we consider the *producer-process-consumer* paradigm (see Figure 4c) where *process* has a data-parallelisation factor up to twice the number of cores, i.e. $\delta = 32$. At all times there are 32 tasks (but only 2δ FIFOs) which process 32 data chunks of $1kB$, produced/consumed by the corresponding tasks. For illustration purposes, as reference architecture we consider a simple multi-core, i.e. one cluster with K cores organised in pairs, and the same number of memory banks ($M = K$). We chose such a simplistic application and architecture as it better illustrates the impact of data-parallelisation than more complicated setups. Results for a NoC-based MPSoC and real-life applications can be found in [4].

As illustrated in Figure 5, by not considering data parallelisation, the estimation of the WCET of the *process* task

[2]This restriction is due to the deployment tool used in Section II.

[3]StreamExplorer from Verimag, available at: http://www-verimag.imag.fr/~poplavko/streamExplorer.html

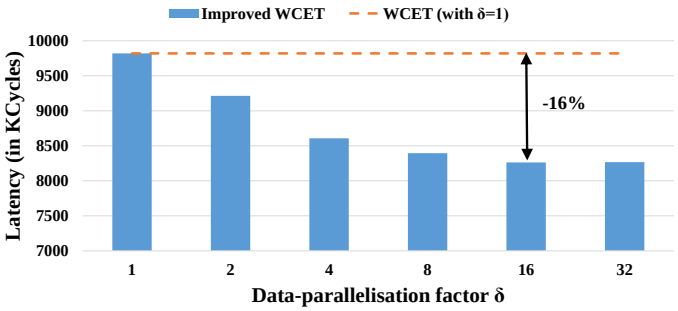


Fig. 5: Latency reduction due to accurate WCET estimation accounting for the data-parallelisation factor δ. Original estimation (dashed line) and corresponding estimation for different values of $\delta = \{1, \ldots, K\}$ for an architecture with $K = 16$ cores at 400 MHz.

is constant at the value for $\delta = 1$, resulting in a constant estimated total application latency.

However, as δ increases, it is possible to distribute FIFOs across more memory banks, thus reducing bus and memory interference. Thus, at each step (except $\delta = 32$), with an interference-aware mapping, the amount of interference can be approximately halved, proportionally reducing the total latency and approaching the WCCT. At $\delta = 32$, since the task count exceeds the memory bank count, the interference cannot be lowered anymore, and the increased task count brings a small constant overhead, therefore the total latency increases slightly.

Overall, we show that by taking into account the data-parallelisation of the application we were able to tighten the guaranteed latency up to 16%, excluding 100% of the potential interferences.

IV. Safe Runtime Adaptation

In Section II we discussed how to acquire a safe and tight deployment $(\mu_E, \mu_M, s, d_{wc})$ for the system model $\mathcal{S} = (\mathcal{V}, \mathcal{E}, d, \sigma)$ (and the corresponding task model $TM = (V, E, d_{wc}, \sigma)$) by excluding sources of interference and exploiting data parallelism. These static WCET estimations d_{wc} are still over-approximated, since worst-case interference rarely happens in reality. Thus, a timed execution of the deployment solution according to schedule function s guarantees safety, but is likely inefficient.

To improve the system performance, we introduce a *runtime* optimisation based on the actual execution times (AET) of tasks. This runtime optimisation resembles self-timed scheduling, but guarantees by construction that no new interference will be introduced, thus preserving safety. This is achieved by transforming the task model TM into a *scheduled task model* $STM = (V, E \cup E_S, d_{wc}, \sigma)$ by enforcing additional *scheduling dependencies* E_S that prevent new interference from happening on any possible executions. Each task v is forced to depend on the previously finished tasks $\{v'\}$ according to the scheduling s, thus maintaining the partial order defined by the schedule s. Intuitively, even if tasks are rescheduled to earlier times, because the AET outperformed the WCET, these dependencies prevent additional overlaps by separating their execution in time [4].

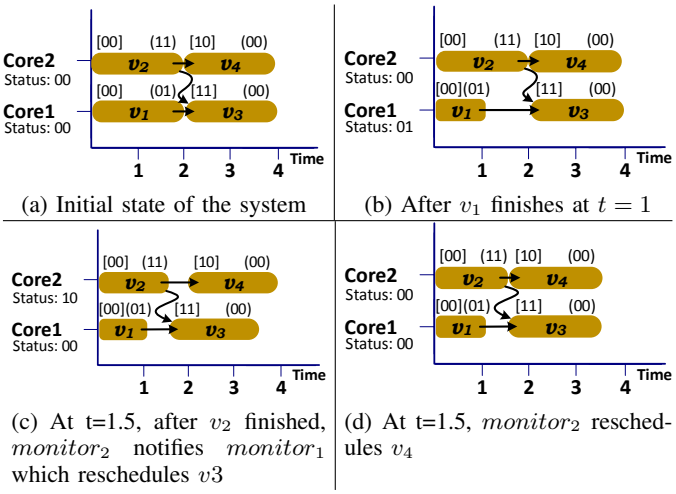


Fig. 6: Example of monitor operation for four tasks on two cores. For each task the *ready mask* is in square brackets. The *dependency vector* is in parentheses and is also illustrated with arrows.

To maximize the performance gains, our approach involves light-weight distributed monitors, one on each core, which are executed every time a task v' finishes, and dynamically adjust the system scheduling. Introducing such monitors can interfere with running tasks and alter their WCET. Therefore, for safety and performance reason such monitors should be be rather simple in order to preserve safety and not undermine performance.

A. Monitor Operation

Since a monitor is invoked every time a task completes, an efficient implementation is imperative. Further, to preserve safety, the monitors must be implemented to have bounded i) execution times and ii) potentially-interfering memory requests. These overheads must be added upfront to the WCCT of every task.

The main purpose of monitors is to yield control to the next task when it is *ready* to execute. If not, a monitor must stall until all dependencies are met. In our implementation, each task holds bit vectors for its outgoing (*dependency vector*) and incoming (*ready mask*) edges. Each bit represents a core of the platform P, on which this edge ends/originates, and is derived from the dependency relations $E \cup E_S$ of the *scheduled task model* and the *task mapping* μ_E. To facilitate the explanation, we consider that these vectors always have the same size $|K| * |X|$ bits, although for tasks that only have intra-cluster dependencies, $|K|$ bits are sufficient.

Each monitor also holds a *status* bit vector, of the same size and initialized at 0, tracking from which cores a dependency has been met. Each monitor can write into the other monitors' *status* vectors, either locally or over the NoC.

After a task v' on core k completes, the k-th monitor updates the *status* of all the cores on the *dependency vector* of v', setting their k-th bit. Subsequently, it notifies (e.g. with an interrupt) the monitors of these cores, in case they are awaiting for the dependencies of their next task to be met. Finally, the

monitor tries to start the next task of the local core k; if not ready, it sets the core in idle mode. This process is illustrated in Figures 6a, 6b and summarised in Algorithm 1.

Algorithm 1: First invocation of a monitor on core k

```
/* Notifies other cores that v' has
finished on core k and executes the
next task v when it becomes ready      */
input: Task v', status array of all cores;
       (status[i][j]: the j-th status bit of the i-th core)
1 Function invokeMonitor(v',status[ ][ ]):
2     v ← v'.next ;
3     dependencies ← v'.dependencies ;
4     for i ← 1 to |K| * |X| do
5         if dependencies[i] then
6             status[i][k] ← 1 ;
7             notifyCore(i) ;
8     startTaskWhenReady(v, status[k]) ;
```

To start the next task, the monitor checks if all the required dependencies are met. If not, it enables incoming notifications (e.g. interrupts) and waits for a notification to recheck. When the task is ready, the monitor disables notifications, and reconfigures its local *status* to 0 (see Figures 6c, 6d and Algorithm 2).

Algorithm 2: Wait until task is ready

```
/* Executes task v when it is ready   */
input: Task v, the status of this core
1 Function startTaskWhenReady(v,status):
2     readyMask ← v.readyMask ;
3     while (status & readyMask) ≠ readyMask do
4         enableNotifications() ;
5         waitNotify() ;
6     disableNotifications() ;
7     status ← 0 ;
8     v.execute() ;
```

B. Runtime Performance

Expanding the results of Section III, we evaluate our runtime optimisation with the *produce-process-consume* reference application in order to outline the impact of interference as data-parallelisation increases in actual executions. In Figure 7, we compare the guaranteed latency, acquired in Section III, with the observed latency (maximum and average) after 100 iterations of the application executed on a Kalray MPPA-256 chip.

We can observe that data parallelisation, as expected, allows for more parallel FIFO mappings onto memory banks, which improves both the guaranteed (tightened WCET) and observed latency. The guaranteed latency decreases proportionally to δ. The observed latency rapidly decreases for δ values between 1 and 2 and then slightly decreases up to $\delta = 4$, when it starts to slightly increase. This difference is explained by the fact that

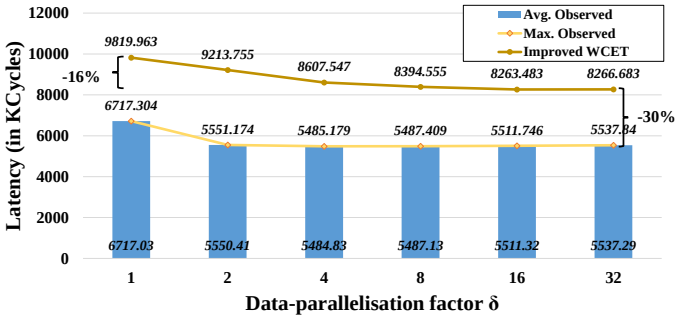


Fig. 7: Comparison of the observed latency vs the guaranteed latency of the reference application for different values of δ deployed on 16 cores at 400MHz.

the WCET analysis expects, for lower values of δ, that more memory requests will experience the worst-case delay. In fact, not all memory requests are delayed and not with the worst-case delays. The slight increase, for $\delta > 4$, of observed latency is due to higher number of tasks and thus higher monitoring overhead. The average and maximum observed latencies are quite close, since all tasks need to perform faster on average, because the *consumer* task waits for all the data.

Overall, by considering the degree of data parallelisation of the application, we were able to improve the guaranteed latency by 16%. By safely adapting execution, we could compress the runtime by an additional 30%, creating slack that could be used for power-saving modes or the execution of lower-criticality tasks (at the end of execution or using interference monitors [6]). In general, we demonstrate that an application mapping flow that is aware of, and appropriately exploits, application data parallelism can exclude potential interferences, leading to tighter and more accurate WCET estimations. The gains from our runtime rescheduling technique also increase when accounting for the data-parallelisation factor of the application.

V. Discussion

While many formal models for safety-critical or mixed-criticality systems exist, most of them do not explicitly account for interferences. Instead either the a-priori knowledge of WCET is assumed, or interference is largely approximated or even safe-guarded using either specialized hardware or sophisticated software and scheduling techniques. The latter is especially true for mixed-criticality systems in order to avoid interference from lower criticality tasks, and for some safety-critical approaches as well.

As outlined in [4], [5], [6] and in Section III, the amount of interference varies significantly and affects guaranteed latency proportionally, but not linearly, especially in the case of data-parallelisable applications. As a result, many theoretical approaches result in inefficient implementations of real-life systems. For example, the authors in [8] present an elaborate scheduling and response time analysis for mixed-criticality systems on many-core platforms, which de-facto prevents interference from lower criticality tasks and does not exploit the possible parallelisation of the applications. Similar con-

clusions can be drawn for the state-of-the-art mixed-criticality approaches as previously noted [7].

On the other end of the spectrum, for safety-critical and mixed-criticality systems, there are approaches that focus on a systems perspective. For example the authors of [6], [12] provide sound scheduling and monitoring techniques for the execution of tasks under the presence of interference. Specifically both propose monitoring techniques to safe-guard the amount of interference a task experiences. Nevertheless, such techniques, operating at a low level, cannot optimise according to the parallelisation factor and can introduce non-negligible interference and overhead in the target system by the constant monitoring of resource usage.

Finally, model-based techniques [9], [11] can be fitting for the problem as they can explicitly model parallelism and provide safe deployments and are suitable for safe runtime adaptation. While they can achieve good runtime performance, the fact that do not model the interference on the underlying architecture results in pessimistic guarantees and under-utilized systems.

VI. Conclusions

In this paper, we presented how theoretical models can be utilised to provide safe and *efficient* solutions to the problem of deploying parallel applications on many-cores. In this paper, we bridge the gap between formal analysis and practical aspects of systems engineering.

We consider that to achieve efficiency it is important to 1) accurately estimate interference delays, 2) consider possible data-parallelisation, as it affects interference, and 3) adapt at runtime with minimal overhead, to preserve safety and efficiency. Our methods improve guaranteed latency by 16%, due to accurate interference estimation, and an additional 30% at runtime with a combined gain of 46% on the reference application. Similar results have been obtained for real-life applications in our previous works [4], [5].

References

[1] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 365–376, ACM, 2010.

[2] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[3] K. Gilles, "The semantics of a simple language for parallel programming," *In Information Processing*, vol. 74, pp. 471–475, 1974.

[4] S. Skalistis and A. Simalatsar, "Near-optimal deployment of dataflow applications on many-core platforms with real-time guarantees," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2017*, pp. 752–757, IEEE, European Design and Automation Association, 2017.

[5] S. Skalistis and A. Simalatsar, "Worst-case execution time analysis for many-core architectures with NoC," in *International Conference on Formal Modeling and Analysis of Timed Systems*, pp. 211–227, Springer, 2016.

[6] A. Kritikakou, C. Rochange, M. Faugère, C. Pagetti, M. Roy, S. Girbal, and D. G. Pérez, "Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, p. 139, ACM, 2014.

[7] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, 2013 (Ninth version: Jan 2017).

[8] G. Giannopoulou, N. Stoimenov, P. Huang, L. Thiele, and B. D. de Dinechin, "Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources," *Real-Time Systems*, pp. 1–51, 2015.

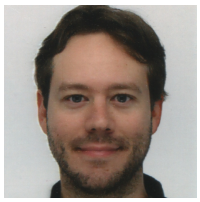
[9] P. Tendulkar, P. Poplavko, I. Galanommatis, and O. Maler, "Many-core scheduling of data parallel applications using SMT solvers," in *Digital System Design (DSD), 2014 17th Euromicro Conference on*, IEEE, 2014.

[10] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pp. 1–6, IEEE, European Design and Automation Association, 2014.

[11] P. Tendulkar, P. Poplavko, and O. Maler, "Symmetry breaking for multi-criteria mapping and scheduling on multicores," in *Formal Modeling and Analysis of Timed Systems*, pp. 228–242, Springer, 2013.

[12] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement," in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pp. 109–118, IEEE, 2014.

Stefanos Skalistis received the Ph.D. degree (2017) from the Swiss Federal Polytechnical School of Lausanne (EPFL), Switzerland. Previously, he has worked in research positions for 2.5 years in AUTh and Coventry University and during his Ph.D. studies in the Rigorourous System Design (RiSD) and Integrated Systems Laboratory (LSI) he focused on real-time systems, correctness-by-construction techniques, scheduling and optimisation.



Federico Angiolini received the Ph.D. degree (2008) from the Department of Electronics and Computer Science from University of Bologna, Italy. His initial research interests included multiprocessor-embedded systems and networks-on-chip, resulting in co-founding the iNoCs Structured Interconnects. Since 2013, he is working on medical imaging and drug delivery platforms at the Swiss Federal Polytechnical School of Lausanne (EPFL).

Alena Simalatsar is a scientific collaborator at HES-SO Valais-Wallis. Previously, she spent six years at EPFL, as a scientific collaborator, conducting research on safety analysis of embedded systems in multiple projects. She received her PhD in Computer Science and Telecommunication Technologies (University of Trento, Italy) in 2009, focusing on system-level analysis methodologies for embedded systems.

Giovanni De Micheli is a professor and director of the Institute of Electrical Engineering at EPFL and the leader of the Nano-Tera.ch program. De Micheli is a recipient of the IEEE Computer Society Harry Goode Award and the European Design and Automation Association (EDAA) Lifetime Achievement Award. He is a member of Academia Europaea and an International Honorary member of the American Academy of Arts and Sciences.