UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Proceedings of the 7th Junior Researcher Workshop on Real-Time Computing: JRWRTC 2013: Sophia Antipolis, France, October 16-18, 2013

Altmeyer, S.

Publication date
2013
Document Version
Final published version

Link to publication

Citation for published version (APA):
Altmeyer, S. (2013). *Proceedings of the 7th Junior Researcher Workshop on Real-Time Computing: JRWRTC 2013: Sophia Antipolis, France, October 16-18, 2013*. Faculty of Science, University of Amsterdam. http://jrwrtc.science.uva.nl/JRWRTC13_proceedings.pdf

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)
Download date:09 Mar 2023

Proceedings of the

7th Junior Researcher Workshop on Real-Time Computing

JRWRTC 2013

http://jrwrtc.science.uva.nl

Sophia Antipolis, France

October 16-18, 2013

Message from the Workshop Chair

Welcome to the 7th Junior Researcher Workshop on Real-Time Computing in Sophia Antipolis. As part of the 21st International Conference on Real-Time and Network Systems (RTNS), the workshop provides junior researchers the opportunity to present their work, share and discuss their ideas and meet with the real-time community in a relaxed forum.

I would like to take the opportunity to express my gratitude to the members of the Program Committee listed below for thoroughly reviewing all papers within a very short time. Furthermore, I thank the General Chairs of the RTNS, Robert De Simone (INRIA Sophia-Antipolis Méditerranée), Michel Auguin (LEAT, Nice) and the local organization committee as well as the Program Chairs, Rob Davis (University of York, UK) and Emmanual Grolleau (LIAS, ISAE-ENSMA, Poitiers), for their help and support in organizing the workshop. Special thanks go to Liliana Cucu-Grosjean (INRIA Rocquencourt, France) for arranging the best paper prize and to INRIA Rocquencourt for sponsoring it.

On behalf of the Program Committee, I wish you a pleasant workshop, hope you will enjoy the presentations and invite you to discuss the presented ideas with the authors during the poster session.

Sebastian Altmeyer (University of Amsterdam)
JRWRTC 2013 Workshop Chair

Program Committee

Mihail Asavoae	Grenoble INP / Verimag, France
Mohamed Bamakhrama	Leiden University, NL
Dakshina Dasari	CISTER Porto, Portugal
Andreas Gustavsson	Mälardalen University, Sweden
Jörg Herter	Saarland University, Germany
Emilien Kofman	INRIA Sophia Antipolis, France
Juri Lelli	Scuola Superiore S.Anna, Pisa, Italy
Will Lunniss	University of York, United Kingdom
Arno Luppold	Ulm University, Germany
Dorin Maxim	INRIA Nancy - Grand Est, France
Mitra Nasri	TU Kaiserslautern, Germany
Gurulingesh Raravi	CISTER Porto, Portugal
Nicolas Serna	LEAT Nice, France
Yassine Ouhammou	LIAS/ISAE-ENSMA, France

Table of Contents

Improving the Precision of Approximations in WCET Analysis for Multi-Core Processors

Michael Jacobs (jacobs@cs.uni-saarland.de)
Saarland University, Saarbrücken, Germany

***Abstract*—The worst-case execution time (WCET) analysis for multi-core processors is a challenge. An explicit consideration of all possible interference effects caused by the shared resources is in many cases combinatorially infeasible. Therefore, approximations are used to reduce this complexity. Current approaches to WCET analysis for multi-core processors are specific to particular ways of approximating the underlying system. Furthermore they are only applicable to rather restricted classes of processors. We identified a common methodology behind existing approaches and formalized it in a unified meta approach. Our meta approach is not restricted to a particular way of approximation. It allows to improve the precision of the obtained WCET bounds by incorporating properties of the system under consideration.**

I. Introduction

Multi-core processors consist of several processor cores, which share common resources such as buses or caches. Their use can reduce the weight, the energy consumption and the production costs of computer systems. Hence, they are likely to also be used for timing-critical applications in the long run [1].

However, resource sharing can have a significant impact on the overall performance of a system [2] because several cores compete for the shared resources. This effect is commonly referred to as *shared resource interference*.

For a timing-critical application it is important that the time needed to deliver the results of its calculations does not exceed a deadline dictated by the physical environment. A time-critical application may consist of several programs that interact. Knowledge about the worst-case execution time (WCET) [3] of each such program allows to verify the timeliness of the overall application. It is safe to replace the WCET of a program by an upper bound on its execution times in this verification step. However, the timeliness of an application can often only be verified if these upper bounds are relatively tight. WCET analyses can be used to derive these upper bounds on the execution times of programs. The execution times of a program depend heavily on possible execution behaviors at the microarchitectural level of the processor that executes the program. From now on, we just use *behaviors* to refer to the execution behaviors at the microarchitectural level of a processor.

Very simple processors allow for a precise WCET analysis by instruction counting or time measurement of a single execution run of the system [4]. Modern processors, however, are too complex to exhaustively simulate or measure the execution times of all possible behaviors. WCET analyses for those processors need to approximate some of the microarchitectural details in order to reduce the inherent complexity [5], [6]. Approximation often comes at the cost of a less tight WCET bound.

The WCET analysis of programs executed on multi-core processors is a special challenge. It needs to take into account all possible interference effects due to resource sharing. A precise consideration of all such effects might in many cases require an exhaustive enumeration of all possible interleavings of accesses to the shared resources by the different processor cores. Such a consideration can be looked upon as combinatorially too complex.

Current approaches to WCET analysis for multi-core processors [7], [8], [9], [10], [11], [12], [13] try to find a level of approximation that avoids this complexity without sacrificing precision too much. Unfortunately, the existing approaches are only applicable to programs executed on very restricted classes of processors. Furthermore, each approach is specific to a particular way of approximating the behavior of the considered system.

In our opinion, a first step toward overcoming these limitations is to identify the common ideas of existing approaches. This will help in designing future approaches in a more generic and uniform way. Our contribution is a meta approach that is not restricted to a particular way of approximation. It allows to improve the precision of the obtained WCET bounds by incorporating properties of the system under consideration.

II. Related Work

The existing approaches to WCET analysis for multi-core processors are derived in formally quite different ways. Yet, the different approaches loosely follow a common two-step methodology.

As a common starting point, all considered approaches assume a level of approximation that does not restrict the amount of shared resource interference. Schranzhofer et al. represent a program executed on a particular core as a sequence of superblocks [7], [8], [9]. A superblock only bounds the number of processor cycles (not blocked at the bus) and the number of bus accesses for a part of the program. Liang et al. assume upper and lower bounds on the number of processor cycles per basic block [10]. These bounds ignore possible penalty cycles induced by the memory hierarchy. The approach of Chattopadhyay et al. [11] bounds the points in time that each program instruction can spend in each pipeline stage of the processor core.

Furthermore, all approaches assume properties of the concrete system under analysis, that provide bounds on the amount of shared resource interference. The approaches considering a shared bus provide bounds on the number of blocked cycles. The intuition behind the bounds stems from the protocol used for bus arbitration (Time Division Multiple Access [7], [11], Round-Robin [12], [13], First-Come-First-Served [12], [8]). The approaches considering a shared cache exploit system properties that guarantee that certain accesses to the shared cache cannot miss [10], [11]. All approaches use these bounds on the shared resource interference to exclude some of the spurious execution behaviors introduced by approximation.

Our meta approach depicts this common methodology in a formal and generic way.

III. Concrete Execution Behavior and Time

We consider a multi-core processor consisting of the set *Cores* of processor cores. For simplicity, we assume that each core only runs one program and that each program may at most be executed once per system run. In the following, we use the term *system* to refer to the combination of the hardware containing the multi-core processor and the software executed on it.

The system may exhibit different execution behaviors depending on its initial state, external input parameters and clock drift effects. Let *Traces* be the set of all execution behaviors of the system. Its superset *Universe* contains the execution behaviors of arbitrary systems.

$$Universe \supseteq Traces$$

Each core C (or the program executed on it) can be assigned an execution time per execution behavior. This time is given by the function et_C.

$$et_C : Universe \rightarrow \mathbb{N} \cup \{\infty\}$$

The WCET of a core C is the maximal execution time for C over all execution behaviors of the considered system.

$$WCET_C = \max_{t \in Traces} et_C(t) \tag{1}$$

IV. Approximation by Abstract Traces

Modern processors usually exhibit too many execution behaviors to allow for an exhaustive consideration of all of them. The set *Traces* is simply too large. Therefore, it is mandatory to introduce some kind of approximation. The goal is to not have to argue separately about each concrete execution behavior.

In our view, an abstract model is given by the tuple $(\widehat{Traces}, \gamma_{trace})$. $\widehat{Traces}$ is the set of abstract traces of the model. The function γ_{trace} maps those abstract traces to subsets of the universe of execution behaviors. Please note that $\mathcal{P}(Universe)$ denotes the power set of this universe of execution behaviors.

$$\gamma_{trace} : \widehat{Traces} \rightarrow \mathcal{P}(Universe)$$

We say that an abstract model $(\widehat{Traces}, \gamma_{trace})$ is an overapproximation of *Traces* iff:

$$\bigcup_{\hat{t} \in \widehat{Traces}} \gamma_{trace}(\hat{t}) \supseteq Traces \tag{2}$$

We assume that for each core C there is an upper bound on its execution times per abstract trace. This bound shall be given by ${}^{UB}et_C$.

$$\begin{aligned} {}^{UB}et_C &: \widehat{Traces} \rightarrow \mathbb{N} \cup \{\infty\} \\ \forall \hat{t} \in \widehat{Traces} &: {}^{UB}et_C(\hat{t}) \geq \max_{t \in \gamma_{trace}(\hat{t})} et_C(t) \end{aligned} \tag{3}$$

From (2) and (3) it follows that the abstract model provides an upper bound to the WCET as defined in (1) by:

$$\max_{\hat{t} \in \widehat{Traces}} {}^{UB}et_C(\hat{t}) \geq WCET_C \tag{4}$$

From now on we only consider abstract models that are overapproximations of *Traces*.

A sound abstract model for a multi-core processor can be derived in a similar way as for a single-core processor by only focussing on one core. This will likely result in very conservative assumptions about the behavior of this core when accessing shared resources as the behavior of shared resources and concurrent cores is not explicitly considered. The baseline approximations of the approaches discussed in Section II follow this paradigm.

V. Infeasible Abstract Traces

The method used to obtain the set of abstract traces (e.g. static analysis) might introduce imprecision. Therefore, there may be abstract traces that only describe spurious execution behavior. We call them *infeasible* abstract traces.

$$\widehat{Infeas} = \{\hat{t} \mid \hat{t} \in \widehat{Traces} \wedge \gamma_{trace}(\hat{t}) \cap Traces = \emptyset\} \tag{5}$$

Correspondingly, we refer to $\widehat{Traces} \setminus \widehat{Infeas}$ as the set of *feasible* abstract traces. In fact, it follows from (5) that the set of feasible abstract traces is an overapproximation of *Traces*.

$$\bigcup_{\hat{t} \in \widehat{Traces} \setminus \widehat{Infeas}} \gamma_{trace}(\hat{t}) \supseteq Traces \tag{6}$$

Based on an abstract model $(\widehat{Traces}, \gamma_{trace})$, which is an overapproximation of *Traces*, we define a set $Deriv_{(\widehat{Traces}, \gamma_{trace})}$ of further abstract models as follows:

$$\begin{aligned} &Deriv_{(\widehat{Traces}, \gamma_{trace})} = \\ &\{(\widehat{Traces}', \gamma_{trace}) \mid \widehat{Traces} \supseteq \widehat{Traces}' \supseteq \widehat{Traces} \setminus \widehat{Infeas}\} \end{aligned} \tag{7}$$

Consider an element $(\widehat{Traces}', \gamma_{trace})$ of set $Deriv_{(\widehat{Traces}, \gamma_{trace})}$. According to (7), $\widehat{Traces}'$ is a subset of $\widehat{Traces}$ that contains at least all feasible abstract traces of $\widehat{Traces}$. It follows from (6) and (7) that each element of $Deriv_{(\widehat{Traces}, \gamma_{trace})}$ is an overapproximation of *Traces*.

$$\begin{aligned} \forall (\widehat{Traces}', \gamma_{trace}) \in Deriv_{(\widehat{Traces}, \gamma_{trace})} &: \\ \bigcup_{\hat{t} \in \widehat{Traces}'} \gamma_{trace}(\hat{t}) &\supseteq Traces \end{aligned} \tag{8}$$

In a similar way as (2) and (3) imply (4), it is a consequence of (8) and (3) that we can calculate an upper bound to the WCET based on any member of $Deriv_{(\widehat{Traces}, \gamma_{trace})}$:

$$\begin{aligned} \forall (\widehat{Traces}', \gamma_{trace}) \in Deriv_{(\widehat{Traces}, \gamma_{trace})} &: \\ \max_{\hat{t} \in \widehat{Traces}'} {}^{UB}et_C(\hat{t}) &\geq WCET_C \end{aligned} \tag{9}$$

As a consequence, we can ignore an arbitrarily chosen set of infeasible abstract traces in an abstract model. A WCET bound based on the remaining abstract traces is still guaranteed to be sound.

The calculation of WCET bounds is based on upper bounds on the execution times per abstract trace (3). If an abstract model makes conservative assumptions about the behavior at the shared resources, some infeasible abstract traces might assume an amount of shared resource interference that exceeds the maximum possible amount for the concrete system. As upper bounds on the execution times of such infeasible abstract traces are likely to be very pessimistic, ignoring those abstract traces—as in (9)—might improve the tightness of the resulting WCET bound significantly.

However, it depends heavily on the abstract model $(\widehat{Traces}, \gamma_{trace})$ and the upper bounds on the execution times per abstract trace if the WCET bound can be tightened by leaving out some infeasible abstract traces. Consider the particular case that the calculation of the WCET bound is dominated by an infeasible abstract trace. Further assume that each feasible abstract trace has an upper bound on its execution times that is strictly smaller than the calculated WCET bound. Then we can obtain a strictly smaller WCET bound by basing its calculation only on the feasible abstract traces. In fact, this proves that the precision of the WCET bound can be improved by pruning infeasible abstract traces.

VI. System Properties

We assume properties to be boolean predicates on execution behaviors. System properties are properties that hold for each execution behavior of a concrete system. The existence of a bound on the shared resource interference may for example be a system property. Let *Prop* be a set of properties of the system under consideration:

$$\begin{aligned} Prop &= \{P_1, \ldots, P_{\#Prop}\} \\ \forall t \in Traces &: \forall P_i \in Prop : P_i(t) \end{aligned} \tag{10}$$

We want to use these system properties to detect some infeasible abstract traces. But so far, they only argue about execution behaviors of the concrete system. Therefore, we need to *lift* them to abstract traces. This means, we need to find $\widehat{P}_i$ such that:

$$\begin{aligned}&\forall \hat{t} \in \widehat{\mathit{Traces}}: \\ &\quad [\,\exists t \in \gamma_{\mathit{trace}}(\hat{t}) : P_i(t)\,] \Rightarrow \widehat{P}_i(\hat{t})\end{aligned} \tag{11}$$

The intuition behind that requirement gets more clear if we have a look at what it means if $\widehat{P}_i$ does not hold for an abstract trace $\hat{t} \in \widehat{\mathit{Traces}}$:

$$\begin{aligned}&\neg \widehat{P}_i(\hat{t}) \\ &\underset{(11)}{\Rightarrow} \forall t \in \gamma_{\mathit{trace}}(\hat{t}) : \neg P_i(t) \\ &\underset{(10)}{\Rightarrow} \gamma_{\mathit{trace}}(\hat{t}) \cap \mathit{Traces} = \emptyset \\ &\underset{(5)}{\Leftrightarrow} \hat{t} \in \widehat{\mathit{Infeas}}\end{aligned} \tag{12}$$

So if a lifted property does not hold for an abstract trace, this means that the abstract trace is infeasible. From now on, the lifted version of any system property shall be identified by the name of the system property with an additional hat on top.

VII. Property Lifting Example

The following example will illustrate how we can find a $\widehat{P}_i(\hat{t})$ satisfying (11) without using $\gamma_{\mathit{trace}}(\hat{t})$ directly, which is mandatory for an efficient use of an abstract model.

Assume that we have an upper bound on the number of bus accesses performed by a particular processor core C per abstract trace.

$$\begin{aligned}&\forall \hat{t} \in \widehat{\mathit{Traces}}: \\ &\quad \forall t \in \gamma_{\mathit{trace}}(\hat{t}): \\ &\qquad {}^{UB}\#\mathit{accesses}_C(\hat{t}) \geq \#\mathit{accesses}_C(t)\end{aligned} \tag{a}$$

We only use γ_{trace} to argue about the soundness of the bounds. But we assume that each bound is given by a preceding analysis in the same way as the corresponding abstract trace is.

In addition, we assume to have a lower bound on the number of cycles that core C is blocked at a shared bus per abstract trace.

$$\begin{aligned}&\forall \hat{t} \in \widehat{\mathit{Traces}}: \\ &\quad \forall t \in \gamma_{\mathit{trace}}(\hat{t}): \\ &\qquad {}^{LB}\#\mathit{blockedCycles}_C(\hat{t}) \leq \#\mathit{blockedCycles}_C(t)\end{aligned} \tag{b}$$

Now assume that the concrete system we consider uses a Round-Robin policy to arbitrate its shared bus. Therefore, all its execution behaviors shall fulfill the property P_{rr}:

$$\begin{aligned}P_{rr}(t) \Leftrightarrow [\,&\#\mathit{blockedCycles}_C(t) \\ &\leq \#\mathit{accesses}_C(t) \cdot (\#\mathit{Cores} - 1) \\ &\quad \cdot \mathit{maxCyclesPerAccess}\,]\end{aligned} \tag{c}$$

The intuition behind this property (implicitly assumed in [12]) is that with Round-Robin arbitration, each concurrent core (there are $\#\mathit{Cores} - 1$ of them) can at most perform one access to the bus before an access of core C is granted. Together with an upper bound on the number of cycles that a granted bus access can at most take to complete on the concrete system, we arrive at an upper bound on the number of cycles that any access of core C can be blocked at the bus. Knowledge about how many accesses to the bus are performed by core C allows us to bound the overall amount of bus blocking experienced by core C in a particular execution behavior.

We can safely lift P_{rr} to abstract traces in a way that satisfies (11) by applying (a) and (b):

$$\begin{aligned}&\forall \hat{t} \in \widehat{\mathit{Traces}}: \\ &\quad \exists t \in \gamma_{\mathit{trace}}(\hat{t}) : P_{rr}(t) \\ &\underset{(c)}{\Leftrightarrow} \exists t \in \gamma_{\mathit{trace}}(\hat{t}): \\ &\qquad \#\mathit{blockedCycles}_C(t) \\ &\qquad\quad \leq \#\mathit{accesses}_C(t) \cdot (\#\mathit{Cores} - 1) \\ &\qquad\qquad \cdot \mathit{maxCyclesPerAccess} \\ &\underset{(a)\,(b)}{\Rightarrow} {}^{LB}\#\mathit{blockedCycles}_C(\hat{t}) \\ &\qquad \leq {}^{UB}\#\mathit{accesses}_C(\hat{t}) \cdot (\#\mathit{Cores} - 1) \\ &\qquad\quad \cdot \mathit{maxCyclesPerAccess} \\ &\Leftrightarrow: \widehat{P_{rr}}(\hat{t})\end{aligned} \tag{d}$$

$\widehat{P_{rr}}$ as defined in (d) clearly satisfies the soundness criterion (11) for lifted properties. According to (12) any abstract trace $\hat{t}$ with $\neg\widehat{P_{rr}}(\hat{t})$ can safely be considered as infeasible.

VIII. Improving the Approximation

We define a compound property $\widehat{P}$ for abstract traces to be the conjunction over the lifted versions of the considered system properties.

$$\begin{aligned}&\forall \hat{t} \in \widehat{\mathit{Traces}}: \\ &\quad \widehat{P}(\hat{t}) \Leftrightarrow \forall P_i \in \mathit{Prop} : \widehat{P}_i(\hat{t})\end{aligned} \tag{13}$$

If $\widehat{P}$ does not hold for an abstract trace $\hat{t}$ then this means that $\hat{t}$ is infeasible:

$$\begin{aligned}&\neg \widehat{P}(\hat{t}) \\ &\underset{(13)}{\Leftrightarrow} \exists P_i \in \mathit{Prop} : \neg \widehat{P}_i(\hat{t}) \\ &\underset{(12)}{\Rightarrow} \hat{t} \in \widehat{\mathit{Infeas}}\end{aligned} \tag{14}$$

We can use $\widehat{P}$ to define an alternative set $\widehat{\mathit{LessTraces}}$ of abstract traces based on $\widehat{\mathit{Traces}}$:

$$\widehat{\mathit{LessTraces}} = \{\hat{t} \mid \hat{t} \in \widehat{\mathit{Traces}} \wedge \widehat{P}(\hat{t})\} \tag{15}$$

$\widehat{\mathit{LessTraces}}$ is the subset of abstract traces in $\widehat{\mathit{Traces}}$ that cannot be classified as infeasible by any of the $\widehat{P}_i$. It follows from (7), (14) and (15) that $(\widehat{\mathit{LessTraces}}, \gamma_{\mathit{trace}})$ is a member of $\mathit{Deriv}_{(\widehat{\mathit{Traces}}, \gamma_{\mathit{trace}})}$.

$$(\widehat{\mathit{LessTraces}}, \gamma_{\mathit{trace}}) \in \mathit{Deriv}_{(\widehat{\mathit{Traces}}, \gamma_{\mathit{trace}})} \tag{16}$$

As a consequence of (9) and (16), we can derive a sound WCET bound from $(\widehat{\mathit{LessTraces}}, \gamma_{\mathit{trace}})$:

$$\max_{\hat{t} \in \widehat{\mathit{LessTraces}}} {}^{UB}\mathit{et}_C(\hat{t}) \geq \mathit{WCET}_C \tag{17}$$

$(\widehat{\mathit{LessTraces}}, \gamma_{\mathit{trace}})$ can improve the precision, as $\widehat{\mathit{LessTraces}}$ potentially prunes some of the infeasible abstract traces still included in $\widehat{\mathit{Traces}}$. In that context, $(\widehat{\mathit{Traces}}, \gamma_{\mathit{trace}})$ is referred to as *baseline abstract model* as it is the starting point for further improvements of precision.

IX. Reducing the Simplifying Assumptions

So far, we assume that each core only runs one program and that each program may at most be executed once per system run. Please note that these restrictions are not inherent to our meta approach. They are ment to facilitate the focus on the essential ideas.

Our meta approach can easily be extended to support several programs per processor core by introducing a set *Programs* of program identifiers. Such an extension will assume the existence of et_{Prog} and ${}^{UB}et_{Prog}$ for each program $Prog \in Programs$. We define the WCET of program *Prog* as follows:

$$WCET_{Prog} = \max_{t \in Traces} et_{Prog}(t)$$

It is straightforward to derive an upper bound to this WCET based on an abstract model that is an overapproximation of the considered system's behaviors:

$$\max_{\hat{t} \in \widehat{Traces}} {}^{UB}et_{Prog}(\hat{t}) \geq WCET_{Prog}$$

In case a program *Prog* can be executed more than once per system run, it is no longer possible to assign a single execution time per program to each execution behavior. Therefore, we use the helper function $runs_{Prog}$ to extract the different execution runs of *Prog* from an execution behavior. In this context, et_{Prog} assigns an execution time to each execution run of *Prog*. An extended definition of the WCET of *Prog* incorporates the execution runs of *Prog*:

$$WCET_{Prog} = \max_{t \in Traces} \max_{run \in runs_{Prog}(t)} et_{Prog}(run)$$

However, the previous definition of a WCET bound for *Prog* can anyway be reused provided that ${}^{UB}et_{Prog}$ fulfills the following criterion:

$$\forall \hat{t} \in \widehat{Traces} : \\ {}^{UB}et_{Prog}(\hat{t}) \geq \max_{t \in \gamma_{trace}(\hat{t})} \max_{run \in runs_{Prog}(t)} et_{Prog}(run)$$

X. Conceptual Approach and Generality

It should be noticed that we describe a *conceptual approach*. Implementations do not necessarily have to stick to its two-step character of first accumulating the set $\widehat{Traces}$ and then sorting out provably infeasible abstract traces.

Note that we did not restrict the form or structure of abstract traces by any means. Therefore, we expect that our meta approach can be mapped to the different ways of approximation used in WCET analyses.

Furthermore, the use of our meta approach is by no means restricted to a scenario of WCET analysis for multi-core processors. Whenever there exists an abstract model that provides an overapproximation of a system's behavior, our meta approach can serve to further improve the precision by additional properties of the system.

XI. Future Work

We plan to instantiate our meta approach for the aiT WCET analyzer[1]. In that way, we intend to come up with an overall approach that supports a wide range of complex processor core features.

In addition to the instantiation of the meta approach for a powerful baseline abstraction, it will be crucial to find a reasonable set of system properties for each supported processor. Those properties have to bound the shared resource interference in a way such that sufficiently tight WCET bounds can be obtained.

Consider system properties that relate the behavior of one processor core to that of other cores. Such properties are typical for systems that do not provide performance isolation between their cores [12], [8]. If an abstract model only focuses on one processor core, then it has to assume arbitrary spurious behaviors for the other cores. The lifted version of a property relating the behavior of the focused core to that of others would have to pessimistically assume the other cores to behave in a way that the property always holds. Therefore, our meta approach currently only profits from such properties if it is used in combination with a baseline abstract model that argues in detail about several processor cores at the same time. But abstract models that argue in detail about several processor cores can be seen as combinatorially too complex. A goal of future work will be to allow for the use of baseline abstract models that focus on one processor core, but that still enable us to incorporate sound assumptions about the behavior of concurrent processor cores.

[1] http://www.absint.com/ait

XII. Conclusion

We presented a conceptual meta approach to WCET analysis for multi-core processors. It points out a common methodology behind previous approaches. Yet, it does not depend on a particular way of approximating the system under consideration. Bounds on the shared resource interference of the concrete system can uniformly be incorporated to improve the precision of the obtained WCET bound.

Acknowledgment

The author would like to thank Sebastian Hahn and Jan Reineke for many comments and interesting discussions.

References

[1] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *Proceedings of the ninth European Dependable Computing Conference*, 2012, pp. 132–143.

[2] A. Abel *et al.*, "Impact of resource sharing on performance and performance prediction: A survey," in *CONCUR*, 2013, pp. 25–43.

[3] R. Wilhelm *et al.*, "The worst-case execution-time problem — overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[4] P. Puschner, "The single-path approach towards WCET-analysable software," in *Proceedings of the IEEE International Conference on Industrial Technology*, vol. 2, 2003, pp. 699–704.

[5] S. Thesing, "Safe and precise WCET determination by abstract interpretation of pipeline models," Ph.D. dissertation, 2004.

[6] X. Li *et al.*, "Modeling out-of-order processors for WCET analysis," *Real-Time Syst.*, vol. 34, no. 3, pp. 195–227, Nov. 2006.

[7] A. Schranzhofer *et al.*, "Timing analysis for TDMA arbitration in resource sharing systems," in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, 2010, pp. 215–224.

[8] R. Pellizzoni *et al.*, "Worst case delay analysis for memory interference in multicore systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 741–746.

[9] A. Schranzhofer *et al.*, "Timing analysis for resource access interference on adaptive resource arbiters," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, 2011, pp. 213–222.

[10] Y. Liang *et al.*, "Timing analysis of concurrent programs running on shared cache multi-cores," *Real-Time Systems*, vol. 48, pp. 638–680, 2012.

[11] S. Chattopadhyay *et al.*, "A unified WCET analysis framework for multi-core platforms," in *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2012, pp. 99–108.

[12] R. Pellizzoni and M. Caccamo, "Impact of peripheral-processor interference on WCET analysis of real-time embedded systems," *IEEE Transactions on Computers*, vol. 59, pp. 400–415, 2010.

[13] D. Dasari *et al.*, "WCET analysis considering contention on memory bus in COTS-based multicores," in *Proceedings of the 16th IEEE Conference on Emerging Technologies Factory Automation*, 2011, pp. 1–4.

An Optimal Design Flow for Hard Real-Time Streaming Systems

Mohamed A. Bamakhrama, Jiali Teddy Zhai, and Todor Stefanov
Leiden Institute of Advanced Computer Science
Leiden University, Leiden, Netherlands
Email: m.a.m.bamakhrama@liacs.leidenuniv.nl

***Abstract*—In this paper, we address the problem of automated design of hard real-time embedded streaming systems. To this end, we introduce the notion of *optimal* design flow. An optimal design flow is one that accepts, as input, a set of hard real-time streaming programs, and then produces in a fully automated manner, as output, the final system implementation, which provably satisfies the timing constraints of the input programs. We propose a realization of such an optimal design flow and implement it. This implementation is called the Daedalus^{RT} design flow and it is available for download, as an open-source framework, from http://daedalus.liacs.nl/.**

I. Introduction

The design of modern embedded streaming systems is a difficult task. The difficulty of this task stems from the fact that multiple functionalities have to be implemented on a single system while meeting stringent timing constraints. In order to tackle this difficult task, modern embedded streaming systems are often realized using Multiprocessor System-on-Chip (MPSoC) technology [1]. In MPSoCs, multiple processors, hardware accelerators, and hardware peripherals are integrated into a single silicon chip. Realizing modern embedded streaming programs on MPSoCs entails several challenges. The first challenge is **expressing the parallelism** in the programs in a way such that (1) we exploit efficiently the multiple processors found in MPSoCs, and (2) we can reason analytically about the performance of the programs. Streaming programs are usually specified at an algorithmic level using a high-level sequential language such as C or MATLAB. Once the correctness of these sequential specifications is verified, they are passed to subsequent design stages. Therefore, it is necessary to parallelize such programs in order to exploit MPSoC platforms efficiently.

The second challenge is **how to allocate and schedule** the programs on the MPSoC such that all the timing constraints of the programs are met. In order to provide such guarantees, the system must use predictable hardware and software. Predictable here means that any HW/SW operation has a bounded worst-case duration. Additionally, the OS scheduler must be capable of enforcing timing isolation between the running programs.

The third challenge is how to design a complex MPSoC with the **least designer effort**. Traditionally, embedded systems have been designed at the level of Register Transfer Level (RTL) for hardware and low-level Application Programmer Interfaces (API) for software. However, as embedded systems move from uniprocessor systems to multiprocessor systems, a shift in the way such systems are designed is also needed.

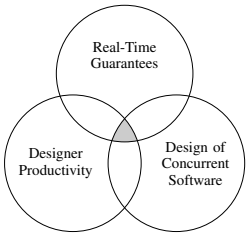


Fig. 1. The challenges involved in designing modern hard real-time multiprocessor streaming systems.

Based on the aforementioned three challenges, we can say that the problem of designing a hard real-time streaming system is an intersection of three sub-problems as shown in Fig. 1. The grey area represents the area to which the aforementioned design problem belongs. In this grey area, the designer must produce a hard real-time multiprocessor system that runs several streaming programs in parallel.

The three sub-problems in Fig. 1 affect each other. For example, the way in which a program is parallelized influences how it will be scheduled during the system run-time. This, in turn, has a direct impact on the timing behavior of the program. Therefore, in order to solve these sub-problems, they must be addressed *simultaneously* by the designer. Solving these sub-problems together can be formulated as follows:

> *Given* a set of hard real-time streaming programs, *devise* a systematic way to parallelize the programs and design, at the right level of abstraction, an MPSoC which runs the parallelized programs *such that* the timing constraints of the programs are guaranteed to be always met during system run-time.

II. Proposed Solution

To address the aforementioned problem, we propose a sequence of steps that the designer should follow in order to arrive at the final MPSoC implementation. These steps constitute a **design flow** (also called **design methodology**). An **optimal** design flow is one which accepts, as input, a set of hard real-time streaming programs, and then produces in a fully automated manner, as output, the final MPSoC implementation. Naturally, an optimal design flow should address the challenges outlined in Section I. Therefore, an optimal design flow should consist of steps that facilitate *maximum design automation* and *correct-by-construction* design. We identify

```
int main() {
  while(1) {
    for(i=1;i<=10;i++) {
      for(j=1;j<=3;j++) {
        src(&img[i][j],&img1[i][j]);

        if(j<=2)
          img[i][j]=f1(img[i][j]);
        else
          img[i][j]=f2(img[i][j]);

        snk(img[i][j],img1[i][j]);
      }
    }
  }
}
```

Listing 1: Example of a SANLP in C

such steps for hard real-time streaming systems and call them **design flow pillars**.

A. Automated Parallelization and Model Construction

Recall that most streaming programs are specified as sequential programs. Most of the execution of these sequential specifications is spent in nested loops [2]. Researchers have investigated several techniques for parallelizing such programs. One particular class of nested loop programs which received a lot of attention is *Static Affine Nested Loop Programs (SANLP)* [3]. This class has been shown to embody a large portion of streaming programs [4]. An example of a valid SANLP is shown in Listing 1.

It has been shown in [3] that a SANLP can be automatically analyzed to construct a parallel version of it. Hence, it is important to utilize this property to relieve the designer from the burden of parallelizing such programs manually. Therefore, the first pillar in an optimal design flow is **automated parallelization**. Given a sequential program, automated parallelization tools analyze the program and construct a parallel version of it. This parallel version of the program exposes the parallelism present in the original sequential program. Several parallelizing compilers were proposed for SANLPs, such as the `PNgen` compiler [5].

Analysis of parallel programs is a tedious task. Therefore, it has been recognized that the designers need to abstract from the actual programs by building *high-level models* of them using Models of Computation (MoC) [6]. Then, these models are used to analyze the program performance under different scheduling and mapping decisions. Such design approach is often called **Model-Based Design (MBD)** which constitutes the second pillar in the proposed design flow. Several MoCs have been proposed in the literature such as Synchronous Data-Flow (SDF, [7]) and its generalization Cyclo-Static Dataflow (CSDF, [8]). Under these models, a program is modeled as a directed graph, where graph nodes represent the tasks in the program and the graph edges represent the data dependencies among the tasks. According to [9], almost all streaming programs can be modeled as SDF graphs. In this work, we choose the CSDF [8] model as the model of computation since it is expressive enough to model most streaming programs as shown in [9].

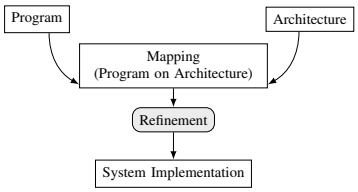


Fig. 2. System-level design of modern embedded systems

B. Real-Time Scheduling Framework

As mentioned earlier in Section I, hard real-time streaming programs have timing constraints that must be always met during the system run-time. To this end, **hard real-time scheduling theory** represents a well-established domain that offers a lot of solutions to the aforementioned issues. It is a mature research area with plenty of results for uniprocessor and multiprocessor systems [10]. Therefore, hard real-time scheduling theory is the third pillar in an optimal design flow.

Hard real-time scheduling theory works in a similar way to model-based design; it abstracts the programs in the form of a *real-time task model*. Such task models impose restrictions on the timing of the program tasks. As a result, it becomes easier to perform timing analysis of the program and reason about its behavior during the design phase. The most famous model is the *real-time periodic* task model proposed by Liu and Layland in 1973 [11]. This model has simple schedulability analysis which led to its wide adoption. In this work, we utilize the periodic task model to analyze the timing behavior of the parallelized programs.

C. System-Level Design and Synthesis

Recall from Section I that modern embedded systems must be designed in a way different than the traditional RTL approach. System-Level Design (SLD, [12]) has emerged as a promising solution to tackle this problem. Under SLD, the system is designed at higher level of abstraction than the RTL level. At system-level, the engineer deals with processors, buses, peripherals, and memories as the primitive blocks that form the system. System-level design abstracts the SoC design by considering it at a high level of abstraction as shown in Fig. 2. In Fig. 2, the system design is a process of *mapping a set of tasks onto a set of processing elements*. Once such a mapping is determined, for example using design space exploration, Electronic System-Level (ESL) synthesis tools [13] provide a (mostly) automated procedure to generate the RTL descriptions for hardware components and the parallel software running on the processors. **System-level design** represents the fourth pillar in the proposed design flow.

III. The Proposed Optimal Design Flow: Daedalus^{RT}

Given the four pillars described earlier, they constitute together the optimal design flow shown in Fig. 3. This design flow is called Daedalus^{RT} [14] and it consists, in total, of six steps. The step number is marked inside a circle in Fig. 3.

Step 1 accepts the SANLPs as input, and then uses the `PNgen` compiler to parallelize them and generate the parallel specification of these input programs. The parallel specification

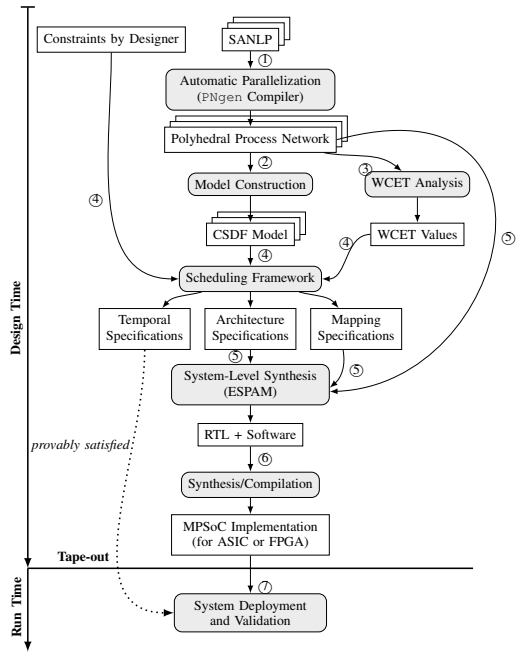


Fig. 3. Overview of the proposed design flow

consists of the Polyhedral Process Network (PPN) representation of the program. PPN is a parallel MoC that is useful for code generation and optimizations. However, it is not suitable for performance analysis. This leads us to the next step.

In *Step 2*, the performance analysis model (i.e., CSDF) is derived automatically from the PPNs generated in step 1. Given a PPN, we propose in [14] an algorithm to derive a CSDF graph that is equivalent to the given PPN.

In *Step 3*, we perform WCET analysis on the parallel specification of the program. WCET analysis can be performed by either static analysis tools or profiling the code on the target MPSoC platform [15].

In *Step 4*, the CSDF models generated in step 2, the WCET values generated in step 3, and the user constraints, which include for example the type of scheduler and other parameters, are fed to the scheduling framework. We propose in [16]–[18] a scheduling framework that derives, for the tasks in the CSDF model, a corresponding real-time periodic task set. The framework computes the parameters of each task (i.e., period, start time, and deadline) and the buffer size of each communication channel such that a valid schedule is guaranteed to exist. After that, the framework performs schedulability analysis based on the scheduler type given by the user. This results in: (i) the architecture specification, which describes how many processors are needed to schedule the programs, and (ii) the mapping specification, which describes the allocation of tasks to processors.

In *Step 5*, the PPNs together with the architecture and mapping specifications are processed by ESPAM [19]. ESPAM is a SLD synthesis tool that supports MPSoC synthesis from PPNs. We have extended ESPAM to support synthesizing the target MPSoC hardware and software. The output from this step is a full MPSoC implementation consisting of the RTL needed to perform low-level synthesis for FPGA or ASIC together with the software running on each processor in the MPSoC.

Step 6 is the last step in the design flow and consists of performing low-level synthesis for FPGA or ASIC backends together with compiling the code for each processor.

TABLE I. TIME NEEDED TO PARALLELIZE AND DERIVE THE CSDF MODEL FOR THE BENCHMARK PROGRAMS ON A LENOVO T500 LAPTOP

Program	# actors	# edges	# lines	Time (seconds)
Filter-bank	69	89	367	1.60
FM radio	28	39	195	0.66
ADI solver	28	167	209	7.26
2D FDT kernel	17	71	144	0.89
2D gauss filter	11	26	75	7.82
Gram-Schmidt	9	20	48	1.85
Regularity detector	8	11	54	2.86

IV. EVALUATION AND RESULTS

In this section, we present the results of empirical evaluation of the Daedalus^{RT} flow explained in Section III. We evaluate the different steps of the flow and demonstrate their effectiveness.

A. Evaluating Automated Parallelization and Model Construction

We evaluate steps 1 and 2 by parallelizing a set of real-life programs and deriving their CSDF models. The used programs are from the PolyBench benchmark [20]. The programs are specified as SANLPs in C and vary in size and complexity. The list of programs together with the time needed to parallelize them and derive their CSDF models is shown in Table I.

The time reported in Table I includes: (1) the time needed by the `PNgen` compiler to parse the C program and generate the parallelized program, (2) the time needed to derive the CSDF model as described in [14]. We see clearly that the first two steps of the proposed flow (i.e., automated parallelization and model construction) are very fast. The fast derivation of the parallelized program and CSDF model relieves the designer from the burden of writing the parallel specifications manually. Moreover, this allows the designer to explore a large number of alternative program specifications in a short period of time.

B. Evaluating Scheduling Framework

The results of evaluating the scheduling framework are described in detail in [16], [17]. We summarize these results here as follows. For 19 real-life programs, scheduling the programs as real-time periodic task sets results in: (1) optimal throughput for 16 programs, and (2) optimal latency for 14 programs. Optimal throughput and latency of a streaming program are those obtained under self-timed scheduling. Under self-timed scheduling, an actor is fired as soon as its input data are available. The aforementioned result shows clearly that periodic scheduling of streaming programs is capable of achieving optimal performance (i.e., throughput and latency) for most programs.

TABLE II. PROGRAMS USED FOR SYSTEM VALIDATION

Program	Description	# tasks
JPEG encoder	Image encoder from raw format to JPEG format	6
JPEG decoder	Image decoder from JPEG format to raw format	2
sobel	Sobel edge-detector filter	5
pipeline	A synthetic program with pipeline topology	4
split-join	A synthetic program with split-join topology	4

C. System Validation

In this step, we validate the systems generated by the Daedalus^{RT} design flow. To this end, we use a set of streaming programs as shown in Table II. First, we synthesize a set of systems, where each system runs a mixture of the programs shown in Table II. Then, the synthesized systems are prototyped on two types of hardware platforms which are: (1) Xilinx ML605 FPGA board, and (2) Avnet ZedBoard with Xilinx Zynq-7000 SoC. We run each synthesized system on actual hardware and monitor its execution to detect deadline misses and/or buffer underflows/overflows. Each synthesized system was validated by running it with real input data for a duration between 1 and 12 hours. For all the synthesized systems, no deadline misses and/or buffer underflow/overflow were detected during the whole validation phase.

V. OPEN ISSUES

In this section, we list the main open issues that we plan to tackle in the future.

1) Support for more expressive MoCs: A more expressive MoC allows a more accurate performance analysis. A first step towards this goal is the work in [21]. The authors in [21] present a scheduling framework similar to ours with support for a MoC called Affine Data-Flow (ADF) graphs, which is a generalization of CSDF. Another option is to support dynamic MoCs which model programs that change their behavior during run-time.

2) Support for programs with cyclic dependencies: Currently, the scheduling framework as proposed in [16], [17] supports only programs with acyclic dependencies. Recently, Benabid et al. [22] showed that any cyclic SDF graph can be scheduled as a set of periodic tasks provided that its back edges contain sufficient amount of initial tokens. Therefore, in theory, it is possible to schedule cyclic CSDF graphs by converting them to SDF graphs. However, it remains an open issue whether or not an analysis technique like the one in [22] can be applied directly on CSDF graphs.

3) Improving the WCET by considering the effect of mapping: During the WCET analysis, we assume that the communication operations take their worst-case latency under a fully congested interconnect. However, such assumption overestimates the WCET value. In a real system, many communication streams are isolated from the others. Therefore, communication operations occur without congestion and they do not take their worst-case latency. Therefore, it is possible to reduce the WCET values if the actual mapping is taken into account.

REFERENCES

[1] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor System-on-Chip (MPSoC) Technology," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 10, pp. 1701–1713, 2008.

[2] B. Franke, "C Compilers and Code Optimization for DSPs," in *Handbook of Signal Processing Systems*, S. S. Bhattacharyya *et al.*, Eds. Springer US, 2010, pp. 575–601.

[3] P. Feautrier, "Dataflow analysis of array and scalar references," *Int. J. Parallel Prog.*, vol. 20, no. 1, pp. 23–53, 1991.

[4] C. Bastoul, "Improving Data Locality in Static Control Programs," Ph.D. dissertation, University Paris 6, Pierre et Marie Curie, France, 2004.

[5] S. Verdoolaege, H. Nikolov, and T. Stefanov, "pn: a tool for improved derivation of process networks," *EURASIP J. on Embedded Systems*, vol. 2007, no. 1, pp. 19–19, 2007.

[6] E. A. Lee and S. Neuendorffer, "Concurrent models of computation for embedded software," *IEE P.-Comput. Dig. T.*, vol. 152, no. 2, pp. 239–250, 2005.

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

[8] G. Bilsen *et al.*, "Cyclo-static dataflow," *IEEE Trans. Signal Process.*, vol. 44, no. 2, pp. 397–408, 1996.

[9] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *Proc. of PACT*, 2010, pp. 365–376.

[10] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, 2011.

[11] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[12] K. Keutzer *et al.*, "System-level design: orthogonalization of concerns and platform-based design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 12, pp. 1523–1543, 2000.

[13] A. Gerstlauer *et al.*, "Electronic System-Level Synthesis Methodologies," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 10, pp. 1517–1530, 2009.

[14] M. A. Bamakhrama *et al.*, "A methodology for automated design of hard-real-time embedded streaming systems," in *Proc. of DATE*, 2012, pp. 941–946.

[15] R. Wilhelm *et al.*, "The worst-case execution-time problem–overview of methods and survey of tools," *ACM T. Embed. Comput. S.*, vol. 7, no. 3, pp. 36:1–36:53, 2008.

[16] M. Bamakhrama and T. Stefanov, "Hard-real-time scheduling of data-dependent tasks in embedded streaming applications," in *Proc. of EMSOFT*, 2011, pp. 195–204.

[17] M. A. Bamakhrama and T. Stefanov, "Managing latency in embedded streaming applications under hard-real-time scheduling," in *Proc. of CODES+ISSS*, 2012, pp. 83–92.

[18] J. T. Zhai, M. A. Bamakhrama, and T. Stefanov, "Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems," in *Proc. of DAC*, 2013, pp. 170:1–170:8.

[19] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and Automated Multiprocessor System Design, Programming, and Implementation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 3, pp. 542–555, 2008.

[20] L.-N. Pouchet, "PolyBench/C: the Polyhedral Benchmark suite," last accessed on: July 22, 2013. [Online]. Available: http://www.cs.ucla.edu/~pouchet/software/polybench/

[21] A. Bouakaz, J.-P. Talpin, and J. Vitek, "Affine Data-Flow Graphs for the Synthesis of Hard Real-Time Applications," in *Proc. of ACSD*, 2012, pp. 183–192.

[22] A. Benabid-Najjar *et al.*, "Periodic Schedules for Bounded Timed Weighted Event Graphs," *IEEE Trans. Autom. Control*, vol. 57, no. 5, pp. 1222–1232, 2012.

Minimizing the cardinality of a real-time task set by automated task clustering

Antoine Bertout, Julien Forget and Richard Olejnik
Laboratoire d'Informatique Fondamentale de Lille
Université Lille 1, France
{antoine.bertout,julien.forget,richard.olejnik}@lifl.fr

ABSTRACT

The objective of this paper is first to properly define the notion of task clustering. This is the process of automatically mapping functionalities (blocks of code corresponding to a high-level feature) with real-time constraints to tasks (or threads). We aim at reducing the number of tasks functionalities are mapped to, while preserving the schedulability of the initial system. Second, our goal is to expose the complexity of the problem and to sketch methods we will propose for solving this problem. We consider independent tasks running on a single processor.

1. INTRODUCTION

Our work falls within the scope of real-time systems programming. Usually, real-time system developers design a system as a set of functionalities with real-time constraints. A functionality is here considered a block of code corresponding to a high-level feature. Implementing such systems requires to map each functionality to a real-time task (thread). On the one hand, the number of those functionalities is quite high. For instance, it ranges from 500 to 1000 in the flight control system of an aircraft or of a space vehicle [6, 10]. On the other hand, a large number of threads implies, a significant time overhead in context switching [23, 13] and an important memory footprint (e.g. task control block, size of the stack, etc.). Thus, the number of tasks supported by embedded real-time operating systems is limited, rarely over one hundred and developers cannot map each functionality to a different task. This mapping is currently mainly performed manually and, given the number of functionalities to process, this work can be tedious and error-prone.

In our work, we address this question from the scheduling point of view. We model a system as a set of tasks with real-time constraints, where each task is characterized by an execution time, an activation period and a deadline, in the same way as Liu and Layland's task model [16]. With respect to this model, functionalities can simply be considered as finer grain tasks, while threads are just coarser tasks. Thus, mapping functionalities to tasks amounts to gathering several tasks into a single one, which we call *task clustering*. Clustering several tasks implies to choose only one deadline for the cluster, which effectively reduces some task deadlines. As a consequence, we have to check that the system schedulability is preserved after the clustering. Our objective is to automate the clustering, so as to reach a minimal task number, while preserving the system schedulability.

Related Work.

In the literature, task clustering is most often studied in the context of distributed systems implementation, where it consists in distributing a set of tasks over a set of computing nodes (processors or cores). This is different from our context, because in the distributed systems context a cluster corresponds to the set of tasks allocated to the same computing resource. For instance, [20, 1] aim at minimizing communications by clustering tasks that communicate a lot. The approaches in [19, 11] cluster tasks based on communications, in order to reduce the system makespan. The number of tasks of the resulting implementation is however not reduced.

Functionality to task mapping is known as runnable-to-task mapping and is identified as a step of the development process in the augmented real-time specification for AUTomotive Open System ARchitecture (AUTOSAR) [5]. This document and [23] also provide guidelines defining under which conditions runnables can be mapped to the same tasks. Authors in [26] propose an automated mapping in that context, but that work is restricted to functionalities that have deadlines equal to their periods. In [7, 18], the authors study the multi-task implementation of multi-periodic synchronous programs and must allocate the different elements of the program to tasks. The clustering is out of the scope of [18], while the heuristic proposed in [7] is very specific to the language structure.

In [22], authors aim at reducing the number of tasks in order to reduce the complexity of the scheduling problem. However, they only focus on functional requirements to group tasks, without considering timing constraints.

This research.

The number of possible clusterings of a task set is equal to the number of partitions of the set, which is close to the *Bell number* [21]. The Bell number is exponential with respect to the cardinality of the set. Thus, given the huge number of possibilities to explore, we motivate the use of a heuristic to tackle the task clustering problem. We also study the schedulability tests that can be applied to first, check the schedulability of a clustering and second, to constitute a relevant heuristic cost function. For now, we do not consider communications and the execution platform is made up of a single processor. These are strong restrictions, which will be lifted in future work. The aim of the present paper is to properly define the problem and to study it in a simple setting, so as to serve as a basis for future work.

Organization.

The rest of the paper is organized as follows. In Section 2, we describe our clustering model. Section 3 is dedicated to the complexity of the task clustering problem. We adress the question of schedulability in Section 4. We describe the current status and the future work involved in Section 5.

2. PROBLEM DEFINITION

Our model, illustrated in Figure 1, is based on Liu and Layland's model [16]. A system consists of a synchronous (i.e. with offsets

equal to zero) set of real-time tasks $\mathcal{S} = (\{\tau_i(C_i, D_i, T_i)\}_{1 \leq i \leq n})$ where C_i is the worst-case execution time (WCET) of τ_i, T_i is the activation period, D_i is the relative deadline with $D_i \leq T_i$. We denote $\tau_{i.k}$ the $(k+1)^{th}$ $(k \geq 0)$ instance, or *job*, of τ_i. The job $\tau_{i.k}$ is released at time $o_{i.k} = kT_i$. Every job $\tau_{i.k}$ must be completed before its absolute deadline $d_{i.k} = o_{i.k} + D_i$

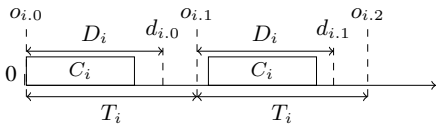


Figure 1: Task Diagram.

2.1 Scheduling

In this paper, we focus on priority-based scheduling policies, either fixed-job with Earliest Deadline First [16](EDF) or fixed-task priority policies with Deadline Monotonic [14](DM).

Let $\mathcal{J}$ denote an infinite set of job, i.e., $\mathcal{J} = \{\tau_{i.k}, 1 \leq i \leq n, k \in \mathbb{N}\}$. Given a priority assignment Φ where 0 is the lowest priority, we define two functions $s_\Phi, e_\Phi : \mathcal{J} \rightarrow \mathbb{N}$, where $s_\Phi(\tau_{i.k})$ is the start time and $e_\Phi(\tau_{i.k})$ is the completion time of $\tau_{i.k}$ in the schedule produced by Φ.

DEFINITION 1. *Let $\mathcal{S} = (\{\tau_i\}_{1 \leq i \leq n})$ be a task set and Φ be a priority assignment. $\mathcal{S}$ is* schedulable *under Φ if and only if: $\forall \tau_{i.k}, e_\Phi(\tau_{i.k}) \leq d_{i.k} \wedge s_\Phi(\tau_{i.k}) \geq o_{i.k}$*

In the sequel, we will also rely on the notion of *laxity*.

DEFINITION 2. *Laxity L (or* slack time*) indicates the maximum delay that can be taken by the task without exceeding its deadline: $L_i = D_i - C_i$.*

2.2 Clustering

Clustering τ_i and τ_j, where $D_i \leq D_j$, produces a cluster τ_{ij} with the following parameters:

$$C_{ij} = C_i + C_j$$

$$T_{ij} = T_i = T_j$$

$$D_{ij} = D_i$$

The cluster deadline is the shortest of the two tasks. Taking the minimum deadline ensures we respect both initial deadlines, even though the constraints will be, in general, more stringent than the initial constraints.

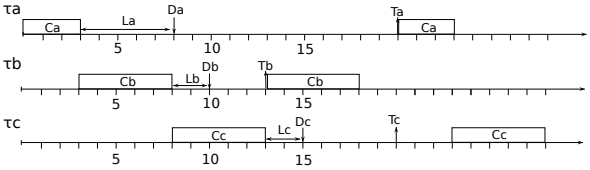
DEFINITION 3. *Let $\mathcal{S} = (\{\tau_i\}_{1 \leq i \leq n})$ be a task set and τ_x and τ_y be two tasks of $\mathcal{S}$ such that $D_x \leq D_y$. We say that τ_{xy} is a* valid cluster *if and only if:*

1. $T_x = T_y$
2. $L_x \geq C_y$
3. *The task set obtained after clustering is schedulable*

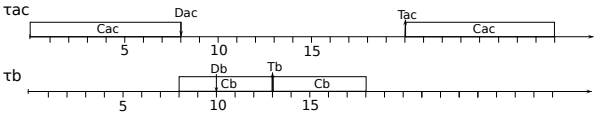
In industrial practices, functionalities of different periods are sometimes mapped together, especially when these functionalities interact a lot, to minimize communication as explained in [24]. This possibility makes the clustering more complex because it requires to manage scheduling inside a cluster. For this reason, we do not deal with this option in this paper. Nevertheless, we could relax this assumption via, e.g., hierarchical scheduling [15].

The laxity test is just an optimization. It is redundant with the schedulability test but it is simpler to check (constant time). Laxity is depicted in Subfigure 2(a).

A schedulable system might become non schedulable after clustering, as illustrated in Figure 2. Indeed, we notice in Subfigure 2(b) that the task τ_b misses its first deadline after the clustering of tasks τ_a and τ_c. Thus, we must check the resulting task set schedulability after clustering.



(a) Initial schedulable system of tasks τ_a,τ_b and τ_c under DM.



(b) Resulting unschedulable system after clustering of tasks τ_a and τ_c.

Figure 2: Influence of task clustering on system schedulability.

3. TASK CLUSTERING COMPLEXITY

We aim in this section at emphasizing the complexity of task clustering, which is related to the search space and to the schedulability test applied.

3.1 Search space

Our problem consists in finding a partition of the task set that is schedulable and with a minimum number of subsets. A partition of a set $\mathcal{X}$ is a set of nonempty subsets of $\mathcal{X}$ such that every element n in $\mathcal{X}$ is in exactly one of these subsets. The number of partitions of a set is the Bell number [21]. The Bell number is exponential with respect to the size of $\mathcal{X}$ and can be computed by the following recurrence relation:

$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$ with $B_0 = 1$

To give a better idea of the size of the search, notice that for instance, $B_{500} \simeq 10^{844}$.

To be more precise, as we only cluster tasks with identical periods, the search space can be restricted to $\prod_{i=0}^{m} B_{n_i}$ where B_{n_i} is the Bell number of the set of tasks with period T_i and m is the number of different periods of the whole task set. Nevertheless, this number remains exponential.

A naive approach might be to conduct an exhaustive search among all partitions of the initial task set, e.g. by applying partitions generation algorithms [2, 17], checking schedulability for each partition generated and choosing the partition with the least subsets. Nonetheless, our first experimentations show that, even using simple, non exact linear schedulability tests (presented below), this solution is not achievable due to the exponential number of partitions to explore. For instance, experiments conducted on a 2.3GHz Intel Core i7 quad-core with 4GByte memory, from an initial set of 20 tasks, lead to more than several days of computation. Thus, we propose to limit the exploration, by applying a heuristic.

3.2 Heuristic function

We start from an initial task set where each task is considered a cluster with one element, we gradually try to group more and more clusters together to minimize the cardinality of the task set. At each step, we try to group one cluster with another and we have, among the candidates that fullfilled conditions 1,2 and 3, some more or less good possibilities. This could be illustrated by Figure 3 for example. Then, we must select the best candidate. This can be achieved by a heuristic cost (or evaluation) function that estimates which candidate will the most likely lead to the best clustering. We propose to achieve task clustering using classic heuristics based on cost functions, such as greedy Best-first search (greedy BFS), A* algorithm or simulating annealing (SA).

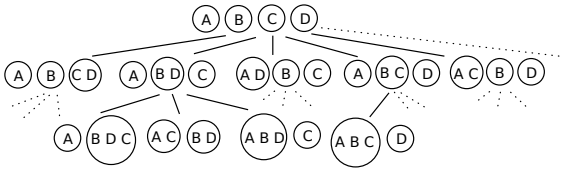


Figure 3: Possible ways to cluster tasks

4. SCHEDULABILITY ANALYSIS

While conditions 1 and 2 adressed earlier can be checked trivially in constant time, condition 3 is more complex. We need a schedulability test to determine a valid task clustering because grouping tasks makes the resulting task set more and more difficult to schedule. Moreover, we need a relevant heuristic cost function to determine the best candidate for the clustering. We want a schedulability test that exhibits some features that might allow us to compare the potential of two task sets. Therefore, in this section, we consider schedulability tests that can be also considered heuristic cost functions.

We present schedulability tests that can be used for clustering under DM and EDF scheduling policies and we detail their ability to be considered a relevant cost function.

A schedulability test is called sufficient if all task sets considered schedulable by the test are actually schedulable. In the same manner, a schedulability test is called necessary if all task sets considered unschedulable by the test are in fact unschedulable. Schedulability tests that are both sufficient and necessary are referred to as exact.

We only consider exact and sufficient tests, thus insuring that the task sets obtained after clustering are schedulable. Indeed, applying sufficient tests means that we might not get the minimum number of clusters but we are sure to still obtain a valid clustering.

4.1 Exact schedulability tests

[8] distinguishes two types of tests: *boolean schedulability tests* and *response time tests*. On the one hand, boolean tests give a boolean answer, determining only whether a task set is schedulable or not, for instance with processor demand analysis (PDA). Thus, they do not exhibit any clear feature that could be considered a heuristic cost function and are not appropriate for our purpose. On the other hand, exact tests based on response time analysis (RTA) provide worst response time for each task and are more suited to be used as cost functions. Indeed, considering a task τ_k with its worst response time denoted R_k, the closer to 1 $\frac{R_k}{D_k}$ is, the less we have margin to group the task τ_k with another. Thus, the sum of each task response time divided by its respective deadline can be used as heuristic cost function. Then, we have a heuristic cost function $h(\mathcal{S})$, such that

$$h(\mathcal{S}) = \sum_{k=0}^{|\mathcal{S}|} \frac{R_k}{D_k}$$

Deadline Monotonic.

RTA [12, 3] of a task τ_i is based on the concept of level-i *busy period*. The level-i busy period is the maximum continuous time interval during which a processor executes tasks of higher or equal priority to the priority of the considered task τ_i, until τ_i finishes its active job. Then, the computation of the worst response time for each task τ_i is based on the length of level-i busy period. RTA for DM can be performed with a pseudo-polynomial time algorithm.

Earliest Deadline First.

Contrary to fixed-task priority (FP) systems, the worst response time is not necessarily found on the first processor busy period in a task set scheduled by EDF [25]. Thus, computing RTA for EDF is more complex and has an exponential complexity.

Even though the RTA for FP has a pseudo-polynomial complexity, early experiments show that the test is quite efficient. The RTA for EDF has an exponential complexity and early experiments seem to show that the test is not practicable (it takes more than several days of computation for 20 tasks).

4.2 Sufficient schedulability conditions

In order to reduce the complexity of the computations, we also considered linear sufficient schedulability tests. Audsley [4] and Devi [9] propose sufficient but not necessary schedulability tests, respectively for DM and EDF in $\mathcal{O}(n)$ complexity. As far as we know, there are no more efficient tests for DM and EDF in linear complexity. The first results show that the test for DM behaves well for clustering and better than that of EDF. Nevertheless, computations with linear test under DM are only 2 times faster than computations with exact RTA test under DM.

Those two sufficient tests actually provide an approximate worst response time for each task. They have a similar form to the exact tests based on RTA. Accordingly, they are also adapted to be used as heuristic cost function.

5. CURRENT STATUS AND FUTURE WORK INVOLVED

We emphasized in this paper that task clustering can not be efficiently achieved by an optimal and exhaustive search but through a heuristic, because of the exponential number of partitions to assess as mentioned in Section 3.1. We explored in this sense the use of sufficient tests and exact tests as heuristic cost functions for DM and EDF.

We are currently working on a heuristic that makes the task clustering feasible. Our preliminary results show that clustering can lead to drastically reducing the number of task, especially when realistic parameters (e.g. with deadlines close to the periods) are used at random task set generation. For instance, we are able to cluster 400 tasks to several dozen in a reasonable time (less than an hour on the machine's configuration cited above) under DM. Results under EDF are less encouraging with high processor utilization factors, probably due to the pessimism of the sufficient test with such settings.

We studied the problem of automatically reducing a large set of independent tasks to a smaller set, while preserving the schedulability of the task set. The current assumption that tasks are indepen-

dent is quite restrictive and will be lifted in future work. Situations where tasks of different periods may be gathered will also be studied.

6. REFERENCES

[1] A. Ahmadinia, C. Bobda, and J. Teich. Temporal task clustering for online placement on reconfigurable hardware. In *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*, pages 359 – 362, Dec. 2003.

[2] J. Arndt. *Matters Computational: Ideas, Algorithms, Source Code*. Springer, 2010.

[3] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[4] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. *Deadline monotonic scheduling*. Citeseer, 1990.

[5] AUTOSAR. *RTE Standard Specifications*.

[6] F. Boniol, P.-E. Hladik, C. Pagetti, F. Aspro, and V. Jégu. A framework for distributing real-time functions. In *Proceedings of the 6th international conference on Formal Modeling and Analysis of Timed Systems*, FORMATS '08, pages 155–169. Springer-Verlag, 2008.

[7] A. Curic. *Implementing Lustre Programs on Distributed Platforms with Real-time Constrains*. PhD thesis, University Joseph Fourier, Grenoble, 2005.

[8] R. I. Davis, A. Zabos, and A. Burns. Efficient exact schedulability tests for fixed priority real-time systems. *Computers, IEEE Transactions on*, 57(9):1261–1276, 2008.

[9] U. Devi. An improved schedulability test for uniprocessor periodic task systems. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 23 – 30, july 2003.

[10] J. Forget. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints*. PhD thesis, Université de Toulouse, 2009.

[11] L. Guodong, C. Daoxu, W. Daming, and Z. Defu. Task clustering and scheduling to multiprocessors with duplication. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 8 pp., Apr. 2003.

[12] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[13] E. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[14] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance evaluation*, 2(4):237–250, 1982.

[15] G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. *Journal of Embedded Computing*, 1(2):257–269, 2005.

[16] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[17] M. Orlov. Efficient generation of set partitions. *Engineering and Computer Sciences, University of Ulm, Tech. Rep*, 2002.

[18] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3):307–338, 2011.

[19] M. Palis, J.-C. Liou, and D. Wei. Task clustering and scheduling for distributed memory parallel architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 7(1):46 –55, Jan. 1996.

[20] K. Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Trans. Parallel Distrib. Syst.*, 6(4):412–420, Apr. 1995.

[21] G.-C. Rota. The number of partitions of a set. *The American Mathematical Monthly*, 71(5):498–504, 1964.

[22] L. Santinelli, W. Puffitsch, C. Pagetti, and F. Boniol. Scheduling with functional and non-functional requirements: the sub-functional approach. *Work-in-Progress Session of ECRTS 2013*, 2:9, 2013.

[23] O. Scheickl and M. Rudorfer. Automotive real time development using a timing-augmented AUTOSAR specification. *Proceedings of ERTS2008*, 4, 2008.

[24] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, and R. Ernst. System level performance analysis for real-time automotive multicore and network architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):979 –992, July 2009.

[25] M. Spuri. Analysis of Deadline Scheduled Real-Time Systems. Research report RR-2772, INRIA, 1996. REFLECS Project.

[26] M. Zhang and Z. Gu. Optimization issues in mapping AUTOSAR components to distributed multithreaded implementations. In *2011 22nd IEEE International Symposium on Rapid System Prototyping (RSP)*, pages 23 –29, May 2011.

Optimism due to serialization in the trajectory approach for switched Ethernet networks

Georges Kemayo, Frédéric Ridouard, Henri Bauer, Pascal Richard
LIAS/ISAE-ENSMA Université de Poitiers
1 avenue Clément Ader – BP40109 – 86961 Futurscope Cedex – France
{georges.kemayo,frederic.ridouard,henri.bauer,pascal.richard}@ensma.fr

Abstract

Many safety-critical applications in avionic systems rely on switched Ethernet networks. A guaranteed upper bound on End-To-End (ETE) communication delays for each flow is mandatory for certification reasons.

Deterministic methods such as the Trajectory Approach have been defined in order to compute a guaranteed upper bound on ETE delay of flows. It has already been proven that in some corner cases, the trajectory approach can be optimistic.

In this paper, we show that the optimization criterion that takes into account the serialization effect can also add further optimism to the computation.

1 Introduction

In this paper, we study high speed packets-network such as, for example, AFDX (Avionics Full DupleX Switched Ethernet)[1] or ATM. We focus on the ETE delays of flows. The flows are constrained by a traffic contract at their ingress point. There is no collision on the physical links. The network elements are designed to ensure that no packet is lost even under maximum congestion conditions. Each network element stores the packet in a buffer using the FIFO (First In, First Out) policy. The configuration of the network is statically defined. Thus each flow (which can be multicast) is identified and statically mapped.

Several approaches have been designed to analyze the ETE delays of flows. The Network Calculus [4, 5, 11] and the Trajectory Approach compute an upper bound of the ETE delay for each flow. We focus on the latter.

This approach has first been introduced by Martin et al. in [10] for FIFO networks. The application on real case scenarios has been proven in [2]. An optimization, considering the serialization of flows is defined in [3]. The sources of the pessimism of the trajectory approach have been analyzed in [8]. Recently, it has been proven that in some corner cases [6], the trajectory approach can be optimistic. Two sources of this optimism have been identified and analyzed.

In this paper, we demonstrate through an example that the optimization criterion taking into account the serialization effect presented in [3] is flawed and can produce optimistic bounds.

The remainder of this paper is organized as follow: the trajectory approach is presented in Section 2. The counter-example is detailed in Section 3. Finally, Section 4 concludes this work.

2 The trajectory approach

The trajectory approach has been defined [10, 9] in order to compute upper bounds on ETE delay of flows in distributed systems context. The trajectory approach model is defined in Section 2.1. The trajectory approach principle is detailed using an example in Section 2.2. Finally, the optimization criterion [3] which is analyzed in this paper, is summarized in Section 2.3.

2.1 Trajectory approach network model

Let $\Gamma = \{v_1, v_2, \ldots, v_n\}$ be a set of n sporadic and unidirectional flows crossing the network. The network is composed of p nodes $\{N_1, N_2, \ldots, N_p\}$ (or network elements). A node consists of a FIFO buffer with 100 Mbps servicing rate. The set of flows crossing a node N_k is denoted Γ_{N_k}. The delay between two consecutive nodes for any packet is constant and denoted L.

A flow v_i is defined by : (1) C_i, the maximum processing time of frames generated by v_i; (2) T_i, the minimum inter-generation time between two consecutive packets of v_i at its network ingress point; (3) $\mathcal{P}_i$, ordered list of crossed nodes from its source node (denoted $first_i$) to the destination node ($last_i$). The cardinality of $\mathcal{P}_i$ is $|\mathcal{P}_i|$.

An important assumption in the trajectory approach is that two flows v_i and v_j meet at most once. In this case, we note $first_{i,j}$ (resp. $last_{i,j}$) their first (resp. last) common node and between these two nodes, they cross the same nodes in the same order (see Figure 1). Finally, when they separate, they do not meet again.

The ETE delay of a flow is defined as the duration between its generation in its source node upon its departure

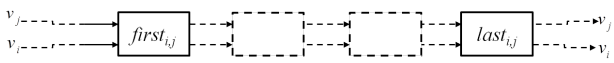


Figure 1. Intersection between two flows

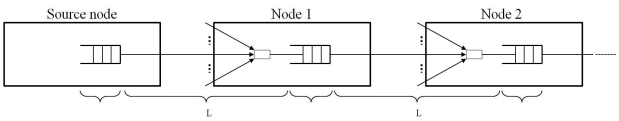


Figure 2. Elements of the ETE delay

from its last node. It corresponds to the sum of the constant technological latencies L and the sum of the waiting times accumulated in each crossed node. The waiting time depends on the backlog present in the buffer at arrival time and therefore, it can suffer varying delays. The ETE delay components are depicted in Figure 2. In this paper, for sake of simplicity, we consider the delay between nodes as negligible (eg. $L = 0\,\mu$s).

2.2 Trajectory approach principle

Let us consider the configuration depicted in Figure 3. It is composed by 8 nodes $\{N_1, \cdots, N_8\}$ and 10 flows $\{v_1, \cdots, v_{10}\}$. The characteristics of the flows are resumed in Table 1.

The upper bound on ETE delay computed with trajectory approach is based on the busy period concept. A busy period [7] is a time interval between two consecutive idle times. An idle time, is a time such as all previously arrived packets have been processed at this time. The trajectory approach analyzes a packet i of v_i generated at time t on its source node and computes an upper bound of its ETE delay.

We denote $W_i^{last_i}(t)$ as the latest starting time on its last node, of the packet i generated by flow v_i at time t (on its source node). The upper bound on the ETE delay of the packet i, generated at time t can be deduced by the following formula: $R_i(t) = W_i^{last_i}(t) + C_i - t$. Finally, all possible generation times t are tested and the worst case is selected as an upper bound. The upper bound of the ETE delay for a packet i generated from flow v_i is given by: $R_i = \max_{t \geq 0} R_i(t)$.

To present the trajectory approach concepts, we analyze packet 7 of flow v_7 following the path $\mathcal{P}_7 = \{N_4, N_5, N_6\}$ from the configuration depicted in Figure 3. An arbitrary scenario of packet 7 generated at time $t = 30\,\mu$s is depicted in Figure 4.

According to [10], the time origin is arbitrarily chosen as the arrival time of the first packet interfering with packet 7 on the source node of the analyzed flow: packet 3 on node N_4. a_i^h denotes the arrival time of packet i on the node h.

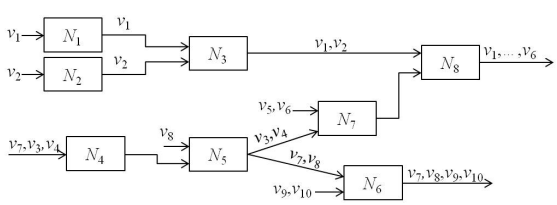


Figure 3. Network configuration.

	$v_1, v_3, \ldots, v_8$	v_2	v_9	v_{10}
C_i	20	20	30	10
T_i	1000	40	1000	1000

Table 1. Characteristics of the flows of the configuration from Figure 3

The delay incurred by packet 7 is related to three busy periods noted bp^{N_4}, bp^{N_5} and bp^{N_6} corresponding to nodes N_4, N_5 and N_6. For each node h, there are two important packets :

- $f(h)$, the first packet executed during bp^h;
- $p(h)$, the "pivot packet": it is the first packet transmitted during bp^h and following the analysed packet i in its next node. In the last node, $p(h)$ is always the packet under study.

In our example, following these notation, the first packet executed during bp^{N_4} is $f(N_4) = 4$ and the pivot packet of the busy period bp^{N_5} is $p(N_5) = 8$.

To determine the latest starting time of packet 7 on its last node N_6, we need to evaluate on each crossed node h:

- the sum of processing time of all packets between $f(h)$ and $p(h)$ (included);
- we then subtract, except for the source node, the difference between the arrival time of packets $f(h)$ and $p(h-1)$ (pivot packet coming from previous node) denoted $\Delta_i^h(t) = (a_{p(h-1)}^h - a_{f(h)}^h)$;
- finally, the processing time of the packet under analysis (packet 7) is subtracted since it is the latest starting time which is computed.

In the example depicted in Figure 4, considering this scenario, the latest starting time of packet 7 on last node N_6, $W_7^{N_6}(30)$, is obtained by adding parts of the three busy periods highlighted in black: $W_7^{N_5}(30) = (20 + 10 + 50) = 80\,\mu$s. The ETE delay of packet 7 generated at time $30\,\mu$s is deduced: $R_7(30) = 80 + 20 - 30 = 70\,\mu$s.

All possible generation times t shall be tested in order to obtain the worst-case ETE delay. But to avoid a combinatorial explosion, an upper bound of the latest starting time and therefore on the delay is computed by the trajectory approach. In [10], some criteria have been established to characterize the worst-case scenario and therefore to determine the upper bound on the ETE delay. Thus, for each crossed node h: (1) the number of packets between $f(h)$ and $p(h)$ is maximized; (2) the term $\Delta_i^h(t) = (a_{p(h-1)}^h - a_{f(h)}^h)$ is minimized. The number of packets between $f(h)$ and $p(h)$ is obtained by maximizing the interference of each met flow. Martin et al. [10]

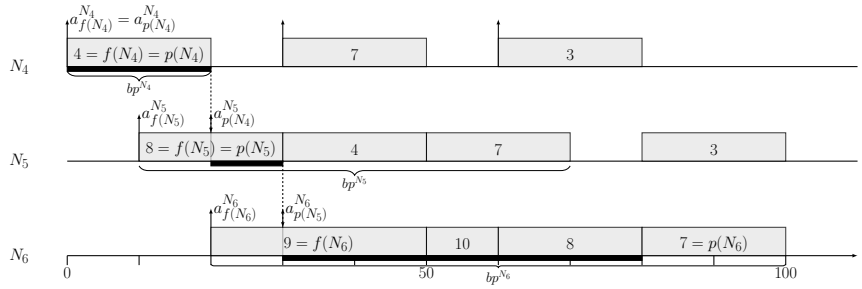


Figure 4. Arbitrary scenario for packet 7

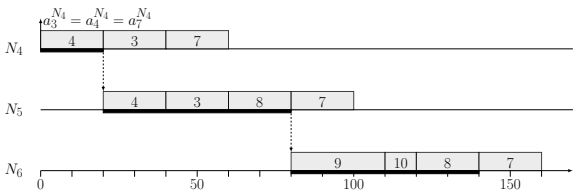


Figure 5. Worst-case scenario for packet 7.

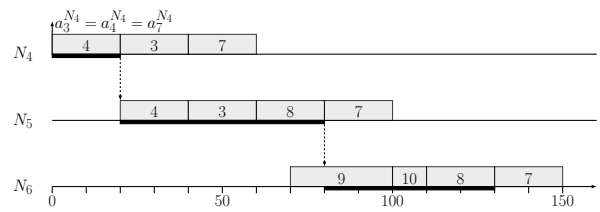


Figure 6. Worst-case scenario considering the serialization effect, for packet 7.

minimize the term $\Delta_i^h(t) = (a^h_{p(h-1)} - a^h_{f(h)})$, for any node h to zero.

Considering the previous criteria, a hand-built scenario leading to the worst-case scenario for packet 7 generated at time $t = 0$ is depicted Figure 5. We obtain: $W_7^{N_5}(0) = (20+60+60) = 140\,\mu$s and the worst-case response time is deduced: $R_7 = 160\,\mu$s

2.3 Optimization considering the serialization effect

But considering the example described in Figure 3 and its flow v_7. A hand-built scenario leading to its worst-case ETE delay is depicted Figure 5. In particular, this scenario takes into account the minimization of term $\Delta_i^h(t)$ to zero [10]. This criterion is pessimistic. In fact, this worst-case scenario considers that packets 9 and 10 arrive simultaneously at time $80\,\mu$s on node N_6. It is impossible since these two packets share the same link to node N_6. Hence they are serialized and they come in sequence.

The Figure 6 presents a worst-case scenario taking into account the serialization effect. The packet 9 arrives on node N_6 at time $70\,\mu$s instead of $80\,\mu$s and the packet 10 arrives simultaneously with the packet $8 = p(N_5)$ at time $80\,\mu$s. Therefore, we obtain $\Delta_7^{N_6}(0) = 10 > 0$. The upper bound on the ETE delay is reduced at $150\,\mu$s instead of $160\,\mu$s.

Taking into account the serialization effect, Bauer et al. [3] define a new minimization criterion to term $\Delta_i^h(t)$.

3 Overestimation due to serialization

In this section, we show that the optimization criterion introduced by Bauer et al. [3] about the term $\Delta_i^h(t)$ taking into account the serialization effect can lead the trajectory approach to be optimistic. Therefore the upper bound on ETE delay computed with the trajectory approach can be flawed.

3.1 Counter-example

We consider the flow v_1 from the configuration detailed Figure 3. The v_1 path is $\mathcal{P}_1 = \{N_1, N_3, N_8\}$. We focus on the packet 1 generated by flow v_1, at time $t = 40\,\mu$s, on its source node N_1. The ETE delay is computed with the trajectory approach, considering the two minimization criterion for the terms $\Delta_1^h(40)$ for each crossed node by v_1.

First, using the basic minimization criterion of Martin et al., $\Delta_1^h(40) = 0$ for any crossed node h, the ETE delay computed for packet 1 generated at time $40\,\mu$s on node N_1 is equal to : $140\,\mu$s. If we use the optimization criterion presented by Bauer et al., we obtain $\Delta_1^{N_8}(40) = \Delta_1^{N_3}(40) = 20$ and therefore, the ETE delay of packet 1 decreases to $100\,\mu$s.

A hand-built scenario focusing on the packet 1 is detailed Figure 7. The packet 1 generated at time $t = 40\,\mu$s on N_1, arrives simultaneously with the packet 2 on node N_3 at time $60\,\mu$s. Another packet from flow v_2, packet $2'$ interferes also with packet 1. Due to the period $T_2 = 40\,\mu$s of v_2, packet $2'$ arrives on node N_3 at time $20\,\mu$s. On the node N_8, the packet 1 meets the packets 3, 4, 5 and 6. Since these packets arrive from the same input link, only the packet 6 can arrive simultaneously with the packet 1 at time $100\,\mu$s. The others packets have arrived previously.

The ETE delay determined visually with this hand-built scenario is $140\,\mu$s. It is equal to the ETE delay computed with the basic criterion from Martin et al. but it is more than the ETE delay computed with the Bauer et al.

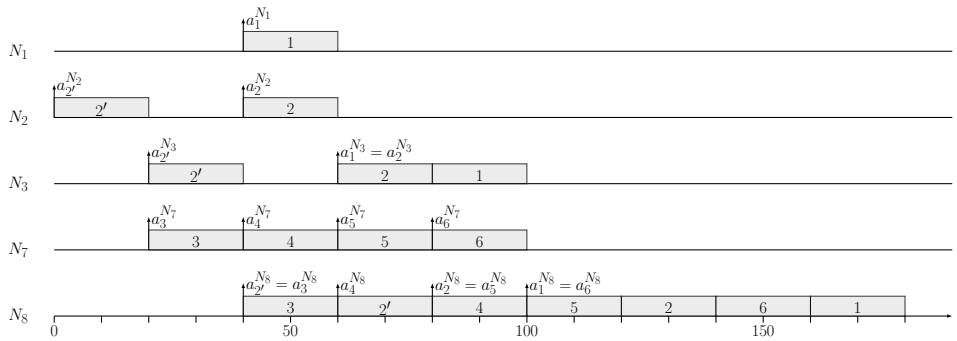


Figure 7. Worst-case ETE delay of packet 1 at $t = 40\,\mu$s

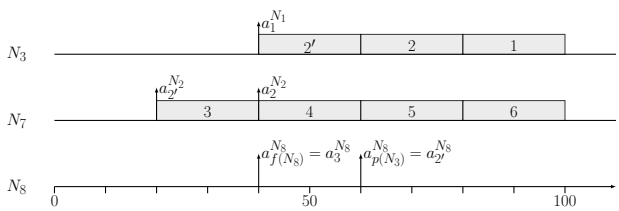


Figure 8. Computation of term $\Delta_1^{N_8}(40)$ according to the optimization criterion proposed by Bauer et al.

optimization criterion which is $100\,\mu$s. We can conclude that the upper bound on the ETE delay using the basic criterion is correct but the optimization criterion can lead to an ETE delay which is no longer an upper bound. Thus, this criterion cannot be used to certificate critical systems.

3.2 Discussion about this optimism

The criterion propose by Bauer et al. [3] can induce optimism in the trajectory approach computation. The terms $\Delta_1^{N_3}(40)$ and $\Delta_1^{N_8}(40)$ are optimist. For example, $\Delta_1^{N_8}(40)$ is minimized by $\mu 20$ using the Bauer et al. optimization criterion taking into account the serialization effect. The computation of $\Delta_1^{N_8}(40)$ is depicted Figure 8. The arrival of packets from each input link (packets 1, 2 and $2'$ for the first link coming from N_3 and packets 3, 4, 5 and 6 for the second link) are delayed at maximum and synchronized between them. This criterion does not respect the period between the packets 2 and $2'$. Therefore, the arrival time of packets $f(N_8) = 3$ and $p(N_3) = 2'$ on node N_8 are separated by $20\,\mu$s which leads to an over estimation of term $\Delta_1^{N_3}(40)$.

4 Conclusion

In this work, we focus on the trajectory approach. This method is used to determine upper bound on ETE delay of flows in distributed systems. We analyze an optimization criterion proposed by Bauer et al. that takes into account the serialization effect between flows sharing the same link. Using a counter-example, we prove that this optimization criterion can induce some optimism in the computed ETE delay.

As future work, we plan to propose a new criterion taking into account the serialization effect for solving the problem exhibited in this short paper.

References

[1] *ARINC 664, Aircraft Data Network, Parts 1,2 7. Technical report, ARINC specification 664.*, 2002-2005.

[2] H. Bauer, J.-L. Scharbarg, and C. Fraboul. Applying trajectory approach to afdx avionics network. *Emerging Technologies and Factory Automation (ETFA 2009)*, pages 1–8, September 2009. Palma de Mallorca.

[3] H. Bauer, J.-L. Scharbarg, and C. Fraboul. Improving the worst-case delay analysis of an afdx network using and optimized trajectory approach. *IEEE Transactions on Industrial Informatics*, 6(4):521–533, November 2010.

[4] J.-Y. L. Boudec and P. Thiran. *Network calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050. Springer Verlag, 2001. ISBN: 3-540-42184-X.

[5] F. Frances, C. Fraboul, and J. Grieu. Using network calculus to optimize the afdx network. In *Embbeded Real-time and Systems (ERTS)*, January 2006.

[6] G. Kemayo, F. Ridouard, H. Bauer, and P. Richard. Optimistic problems in the trajectory approach in fifo context. In *18th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA)*, Cagliari, Italy, September 2013. IEEE.

[7] J. Lehoczky. Fixed-priority scheduling of periodic task sets with arbitrary deadlines. *Real-Time Systems Symposium*, pages 201–209, 1990. Lake Buena Vista.

[8] X. Li, J.-L. Scharbarg, and C. Fraboul. Analysis of the pessimism of the trajectory approach for upper bounding end-to-end delay of sporadic flows sharing a switched ethernet network. *International Conference on Real-Time and Network Systems*, pages 149–158, September 2011. Nantes.

[9] S. Martin. *Mastering the time dimension of the quality of service in networks*. PhD thesis, Univ. Paris XII, 2004.

[10] S. Martin and P. Minet. Schedulability analysis of flows scheduled with fifo: application to the expedited forwarding class. Rhodes Island, Greece, april 2006. IEEE International Parallel and Distributed Processing Symposium.

[11] J.-L. Scharbarg, F. Ridouard, and C. Fraboul. A probabilistic analysis of end-to-end delays on an afdx avionic network. *IEEE Transactions on Industrial Informatics*, 5(1):38–49, Feb. 2009.

History-Cognisant Time-Utility-Functions for Scheduling Overloaded Real-Time Control Systems

Florian Kluge, Florian Haas, Mike Gerdes, Theo Ungerer
Department of Computer Science, University of Augsburg, Germany
{kluge,haas,gerdes,ungerer}@informatik.uni-augsburg.de

***Abstract*—Time-utility functions (TUFs) as means for scheduling allow to build flexible real-time systems. TUFs yield utility values for single job executions. We extend this concept to *history-cognisant utility functions (HCUFs)* that assign an overall utility value to a whole task. We aim to use such HCUFs to improve the single-task performance in control systems. We present two extensions of the EDF policy for overloaded firm real-time systems and compare them with related approaches. First promising results are shown.**

I. Introduction

The notion of deadlines as the sole scheduling criterion is often too strict and can lead to inflexible and performance-sacrificing real-time systems. Instead, Jensen et al. [10] propose to use *time value functions (TVFs, also Time Utility Functions, TUFs)* as a basis for scheduling. A TUF represents a task's value that it will contribute to the system if it is completed up to a certain time. A TUF can be based on task deadlines, but may also incorporate other parameters, e.g. the significance a task has for the operation of a system.

Jensen et al.'s work [10] was extended for best effort scheduling [14] and tasks with dependent activities [8]. In the meantime, several heuristic scheduling algorithms have been proposed (e.g. [7], [12], [20]). The notion of time-value is used in scheduling of real-time systems in general (see [19] for an overview, or e.g. [1], [6], [9]) and in the special case of overloaded real-time systems (e.g. [4], [11], [15], [16]). Applications of time-value scheduling can be found in dynamic reconfiguration of systems [5], Ethernet packet scheduling [20] and robotics [2].

To the best of our knowledge, hitherto existing works on TUF-based scheduling only take the possible value of the current task instance into account, but do not care about a task's previous execution behaviour. In our work, we target periodic control loops that can tolerate sporadic deadline misses. If a control algorithm is built robust and executed at a rate high enough, the system can tolerate single iterations to fail, as it is able to recompense for these in the following iterations [17]. Such behaviour has been formalised e.g. in the (m, k)-firm real-time task model [18] and the weakly-hard real-time model [3]. Nevertheless, control loops are most often implemented as periodic tasks with hard deadlines, thus sacrificing performance and flexibility.

Our aim is to exploit the robustness of such control loops and execute them in a more flexible manner. Scheduling decisions influence the behaviour of the control system and can lead to a degradation of its quality. In this paper, we present an approach to distribute such degradations equally over all control tasks in the system at hand. Therefore, we introduce *history-cognisant utility functions (HCUFs)*. A HCUF maps a task's execution history into a single value which can be evaluated by scheduler. We demonstrate how these HCUFs can be used for scheduling overloaded control systems. If an overload situation is detected, we cancel single jobs until all other jobs can meet their deadlines. Using HCUFs as base for the decision which jobs shall be cancelled, we achieve that (1) cancellations are distributed equally over all tasks in the system, and (2) the number of subsequent cancellations affecting one task is reduced. While our approach is currently based on timing parameters solely, it is possible to refine the utility functions for an actual implementation using further metrics that might also allow for an interaction between the control algorithm and the scheduler.

This paper is structured as follows: In section II we review the concept of TUFs and introduce history-cognisant utility functions. A scheduling approach based on HCUFs is presented in section III. Preliminary evaluation results are shown in section IV. We conclude this paper in section V.

II. Utility Functions

Utility functions are applied to jobs that are generated by tasks. A task τ_i generates jobs $j_{i,k}$ at times $a_{i,k}$. Each job has a deadline d_i relative to its activation time $a_{i,k}$ and an absolute deadline $\hat{d}_{i,k} = a_{i,k} + d_i$. The completion time of job $j_{i,k}$ is denoted as $c_{i,k}$. All utility functions map into the utility domain $\mathbb{U} := [0, 1] \cup \{-\infty\}$. 1 represents maximum benefit for the system that can degrade down to 0 meaning no benefit; $-\infty$ stands for a failed job with possibly catastrophic consequences.

A. Traditional Utility Functions

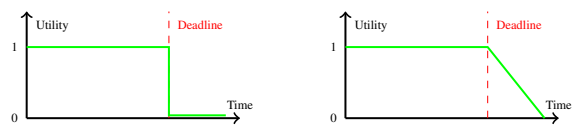
Time-utility functions can be used to assess single job executions. The concept of firm real-time jobs that are worthless once they exceed their deadline can be represented by the following utility function which is also depicted in figure 1(a):

$$u_F(j_{i,k}) = \begin{cases} 1 & c_{i,k} \leq \hat{d}_{i,k} \\ 0 & \text{else} \end{cases} \tag{1}$$

A utility function for soft real-time jobs which are allowed to miss their deadlines can be defined exemplarily in the

following manner (see also figure 1(b)):

$$u_S(j_{i,k}) = \begin{cases} 1 & c_{i,k} \leq \hat{d}_{i,k} \\ 1 - \frac{c_{i,k} - \hat{d}_{i,k}}{d_i} & \hat{d}_{i,k} \leq c_{i,k} \leq \hat{d}_{i,k} + d_i \\ 0 & \text{else} \end{cases} \quad (2)$$



(a) firm real-time (b) soft real-time

Figure 1. Exemplary utility functions

These functions stand only as examples for utility functions. Actual utility functions may have different shapes (see e.g. [10]). Based on such TUFs, we want to estimate the utility of successive job executions.

B. History-Cognisant Utility Functions

A history-cognisant utility function maps the utility values of multiple successive job executions of one task into a single utility value for the task. Insofar, we view a task τ_i as a sequence of jobs $J_i = (j_{i,0}, j_{i,1}, \ldots)$. The notion of *successive jobs* is formalised through connected subsets of sequences:

Definition 1: Let $X = (x_n) = (x_0, x_1, \ldots)$ be a (possibly infinite) sequence. Then $X|_{p,q}$ with $p, q \in \mathbb{N}_0, p < q$ denotes the *connected subset* $(x_p, x_{p+1}, \ldots, x_{q-1}, x_q)$ of X. The whole sequence X can be written as $X|_{0,\infty}$, a tail of the sequence as $X|_{p,\infty}$. Additionally, let

$$\mathcal{S}(X) = \{X|_{p,q} | p, q \in \mathbb{N}_0\} \cup \{X, \emptyset\} \quad (3)$$

be the set of all connected subsets of a sequence X.

To estimate the contribution of a task to the overall system value, we use history-cognisant utility functions U that work on sequences of job executions (with $p \leq q$ and some utility functions u):

Definition 2: A history-cognisant utility function maps the utilities of multiple subsequent job executions $J_i|_{p,q}$ of a task τ_i into a single value that represents the the task's contribution to the system's benefit:

$$\begin{array}{rccl} U: & \mathcal{S}(J_i) & \rightarrow & \mathbb{U} \\ & J_i|_{p,q} & \mapsto & U(u(j_{i,p}), \ldots, u(j_{i,q})) \end{array} \quad (4)$$

The following two examples illustrate concrete definitions of HCUFs:

Example 1: One way to define a concrete HCUF is to calculate the average utility U_A of a subsequence of jobs $J_i|_{p,q}$:

$$U_A(J_i|_{p,q}) = \frac{\sum_{k=p}^{q} u(j_{i,k})}{q - p + 1} \quad (5)$$

Using U_A, all job executions within the window $J_i|p, q$ yield the same influence on a task's utility.

In a robust control system, a single job execution that dates back longer has less influence on a system's current state. We model this behaviour with the following recursive HCUF U_E.

Example 2: Let $U_E(j_{i,k})$ be the utility of task τ_i after the execution of the k-th job $j_{i,k}$. Using some weight $w \in (0, 1)$ and a TUF $u(j_{i,k})$, we define $U_E(j_{i,k})$ as:

$$\begin{array}{rcl} U_E(j_{i,0}) & = & 1 \\ U_E(j_{i,k}) & = & (1 - w)U_i(j_{i,k-1}) + wu(j_{i,k}) \end{array} \quad (6)$$

In our future work we will investigate the influence of the parameter w on an application. Also, we plan to define further HCUFs that e.g. take into account how often a task deviates from a desired behaviour succeedingly.

III. Utility-based Scheduling

Jensen et al. [10] and Locke [14] proposed an extension to the earliest deadline first (EDF) scheduling algorithm [13] to handle possible overload situations. As long as no overload occurs, their best-effort algorithm (in the following called BE) implements the original EDF online policy. If high probability for an overload is detected during runtime, the algorithm modifies an EDF schedule based on the *value density* of each task. The value density for a task τ_i is calculated as V_i/C_i, with V_i being the value of the task, and C_i its processing time. The algorithm removes tasks with lowest value density from the schedule until the overload probability drops below a predefined threshold.

Aldarmi and Burns [1] address one problem of this approach stemming from the static value density (SVD) used in BE: in choosing a task to remove from a schedule, BE does not care whether the task has already started executing. If a running task is chosen for removal, the work it has performed until that point will be lost, and the computing time used up was therefore wasted. Aldarmi and Burns propose to use *dynamic value density (DVD)* as base for scheduling decisions. They calculate a task τ_i's priority $P_i(t)$ at some time t as $P_i(t) = V_i(t)/\bar{C}_i(t)$ with $\bar{C}_i(t)$ being the task's *remaining* execution time. Thus, once a task has started executing, its priority will increase and probability of cancellation decreases.

We notice another problem that can occur with the use of BE: Jobs of tasks with a low value density are cancelled with a higher probability during overload situations. Concerning control applications, such behaviour can be counterproductive: complex control loops with a high execution time might be cancelled more often. In this case, it will be helpful if all tasks suffer a similar degradation, but still are able to provide some guaranteed value to the system.

To address this problem, we propose to use one HCUFs as base for priority calculation to make the scheduling process history-cognisant. Our approach is based on the fact that robust control loops can tolerate single executions to fail. However, such failures must not occur too often. If the scheduler has to perform job cancellations to make a schedule valid, we want to prefer such tasks for cancellation that performed successfully before. Tasks that suffered from cancellations before shall be preferred for execution.

We modify the BE scheduling approach by using different criteria for removal of jobs from an overloaded schedule. Instead of removing low-value-density jobs, we choose jobs

Table I
TASK SET TS1: GOOD-NATURED WITH 100% UTILISATION THROUGH PERIODIC TASKS

Task	P_i/o_i	C_i	d_i	p_A
p1	6/0	2	6	-
p2	6/2	2	6	-
p3	6/4	2	6	-
s1	5	1	5	0.33

Table II
TASK SET TS2: ILL-CONDITIONED, PERIODIC TASKS CREATE OVERLOAD; $o_i = 0$ FOR ALL TASKS

Task	P_i	C_i	d_i	p_A
p1	15	3	6	-
p2	57	8	13	-
p3	111	9	19	-
p4	42	10	18	-
p5	37	5	19	-
s1	50	5	10	0.4
s2	10	2	5	0.2
s3	100	13	20	0.3

whose tasks have accumulated a maximum utility U until now. This leads to our first history-cognisant EDF extension, which we call HC1 in the following. Additionally, the tasks shall also profit from the findings concerning DVD. Therefore, we also include a job's remaining execution time $\bar{C}(t)$ in our removal metric. This leads to a removal of jobs that have maximum values of $U(t) \cdot \bar{C}(t)$. We call this second history-cognisant EDF extension HC2.

IV. EVALUATION

We have performed preliminary use case evaluations of the HCUFs to compare the four extensions BE, DVD, HC1 and HC2 of the EDF scheduling policy. Any extension becomes active, if an overload situation is detected and removes tasks from the schedule. We are interested, how the different scheduling approaches influence the overall completion rates of the single tasks. Also, we investigate how cancellations are distributed over a single task's life time.

A. Scenario

All scenarios that we investigated so far are based on the following assumptions: time is divided into discrete time steps. If a task τ_i is activated, it generates a job $j_{i,k}$, $k = 0, 1, \ldots$ that must be executed. All tasks τ_i are firm real-time with deadline d_i relative to their activation time. Their utility is calculated according to equation (1). Periodic tasks pi are activated at the beginning of their period P_i, heeding a possibly positive offset o_i. Periodic tasks may have deadlines equal to their period, but can also have shorter ones. Sporadic tasks si may be activated with a certain probability p_A in each time step after their minimum separation time has elapsed. Any task τ_i accumulates utility through a history-cognisant utility function U_i as defined in equation (6). In our simulations we chose $w = 0.5$.

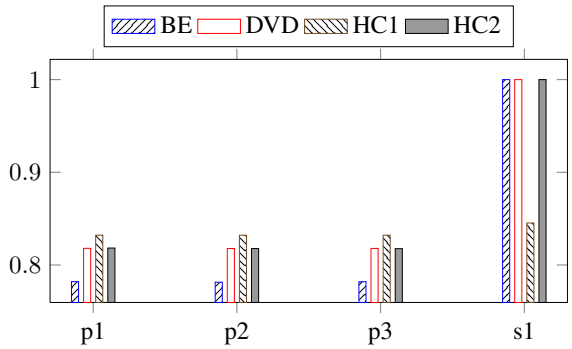


Figure 2. Completion rates of task set 1

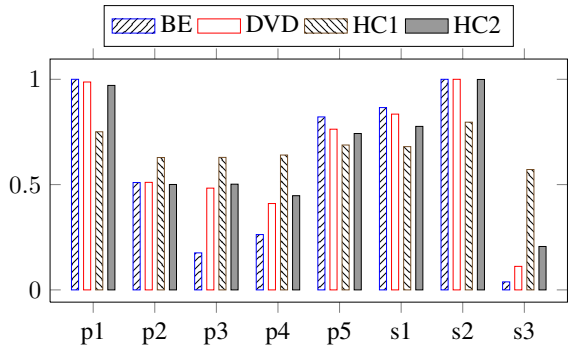


Figure 3. Completion rates of task set 2

So far, we have performed evaluations with several task sets. For reasons of space, we only present the results of two task sets. Task set 1 (table I) is rather well-natured. The periodic tasks create 100% processor utilisation. Any time the sporadic task is activated, the scheduler has to cope with an overload situation. We chose this task set because the overload is well under control and easy comprehensible. Task set 2 (table II) is ill-conditioned. The periodic tasks alone often lead to overload situations that can be aggravated by additional sporadic tasks. The aim of this task set is to estimate, how the different scheduling approaches perform under very hard conditions. For any sporadic task, the number recorded in the P_i column indicates the task's minimum separation time between to subsequent activations.

In our simulations, jobs are generated and executed for one million time steps. After this time no new jobs are generated, and only those already activated are finished.

B. Completion Rates

In task set 1, the periodic tasks generate a basic load of 100%. The sporadic task (s1) gains a completion rate of 1 under BE, DVD and HC2 (see fig. 2). This happens at the cost of the periodic tasks. However, their degradation depends on the concrete policy. The completion rates under BE are lower than those under DVD and HC2. This is due to the fact that DVD and HC2 heed the remaining execution times of tasks that might be cancelled. Both approaches prefer those tasks for cancellation that have not yet started execution. HC1 achieves a nearly equal distribution of task cancellations. In task set

Table III
MAXIMUM LENGTH OF SUBSEQUENT CANCELLATIONS FOR TASK SET 2

	BE	DVD	HC1	HC2
p1	0	1	3	2
p2	7	9	3	6
p3	34	10	2	7
p4	22	13	3	9
p5	5	5	2	4
s1	3	5	3	4
s2	0	0	3	1
s3	134	73	3	48

2, BE and DVD discriminate against tasks that have longer execution times (see fig. 3). HC1 nearly aligns the completion rates of all tasks. Due to the stronger irregularity of the task set, it cannot achieve this completely. HC2 again behaves similar to DVD, but can weaken the discrimination in some places.

All in all, HC1 achieves the best alignment of completion rates among the tasks of a task set. This results from HC1 only taking a task's previous completion behaviour into account, if a cancellation is necessary. HC2 performs similar to DVD, as both approaches also regard a job's (remaining) execution time.

C. Subsequent Cancellations

Next, we examine how cancellations are distributed over a task's lifetime. Therefore, we count how often any sequence of n subsequent cancellations for each task occurs during our simulations of task sets 1 and 2. In task set 1, the longest sequence had length 9, occurring under BE, while all other policies lead to only isolated cancellations. The maximum lengths that occurs during the execution of task set 2 are displayed in table III. Again, longest cancel sequences occurred under BE scheduling, and shortest ones under HC1. Though not fully evaluated yet, these first results indicate that task that are executed using HC1 will most probably not suffer from extensive cancellation sequences.

V. CONCLUSIONS AND FUTURE WORK

We have presented history-cognisant utility functions as an extension of the well-known time-utility functions that can be used for real-time scheduling. We have demonstrated the use of HCUFs through an extension of the EDF scheduling algorithm. The results obtained in our evaluations indicate that the use of apt utility functions can help improving single-task performance in overloaded firm real-time systems. In the future, we will perform extensive evaluations with random task sets to get a more detailed image of the behaviour of our approach. Additionally, we plan to investigate further TUFs and HCUFs. We will compare our approach to other overload schedulers. Especially, we will extend our approach such that it can give the same guarantees as the (m, k)-firm real-time scheduler. Finally, we aim to integrate our HCUFs into current TUF-based schedulers.

REFERENCES

[1] S. A. Aldarmi and A. Burns. Dynamic value-density for scheduling real-time systems. In *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, pages 270–277, 1999.

[2] A. Baums. Indicators of the real time of a mobile autonomous robot. *Automatic Control and Computer Sciences*, 46(6):261–267, 2012.

[3] G. Bernat, A. Burns, and A. Liamosi. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4):308 –321, Apr. 2001.

[4] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. In *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, pages 90–99, 1995.

[5] E. Camponogara, A. B. de Oliveira, and G. Lima. Optimization-Based Dynamic Reconfiguration of Real-Time Schedulers With Support for Stochastic Processor Consumption. *Industrial Informatics, IEEE Transactions on*, 6(4):594–609, 2010.

[6] H. Chen and J. Xia. A Real-Time Task Scheduling Algorithm Based on Dynamic Priority. In *Embedded Software and Systems, 2009. ICESS '09. International Conference on*, pages 431–436, 2009.

[7] K. Chen and P. Muhlethaler. A scheduling algorithm for tasks described by time value function. *Real-Time Systems*, 10:293–312, 1996. 10.1007/BF00383389.

[8] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, Aug. 1990.

[9] W. Ding and R. Guo. Design and Evaluation of Sectional Real-Time Scheduling Algorithms Based on System Load. In *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 14–18, 2008.

[10] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *6th Real-Time Systems Symposium (RTSS '85), December 3-6, 1985, San Diego, California, USA*, pages 112–122, Dec. 1985.

[11] G. Koren and D. Shasha. Dover; an optimal on-line scheduling algorithm for overloaded real-time systems. In *Real-Time Systems Symposium, 1992*, pages 290–299, 1992.

[12] P. Li, H. Wu, B. Ravindran, and E. D. Jensen. A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints. *IEEE Transactions on Computers*, 55(4):454 – 469, Apr. 2006.

[13] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973.

[14] C. D. Locke. *Best-effort decision-making for real-time scheduling*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1986.

[15] P. Mejia-Alvarez, R. Melhem, and D. Mosse. An incremental approach to scheduling during overloads in real-time systems. In *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pages 283–293, 2000.

[16] D. Mosse, M. E. Pollack, and Y. Ronen. Value-density algorithms to handle transient overloads in scheduling. In *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, pages 278–286, 1999.

[17] L. Palopoli, L. Abeni, G. Buttazzo, F. Conticelli, and M. Di Natale. Real-time control system analysis: an integrated approach. In *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pages 131–140, 2000.

[18] P. Ramanathan. Overload management in real-time control applications using (m, k)-firm guarantee. *Parallel and Distributed Systems, IEEE Transactions on*, 10(6):549–559, 1999.

[19] B. Ravindran, E. D. Jensen, and P. Li. On recent advances in time/utility function real-time scheduling and resource management. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 55–60, 2005.

[20] J. Wang and B. Ravindran. Time-utility function-driven switched ethernet: packet scheduling algorithm, implementation, and feasibility analysis. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):119 – 133, Feb. 2004.

Schedule-aware Distribution of Parallel Load in a Mixed Criticality Environment

Marc Bommert
RheinMain University of Applied Sciences, Wiesbaden, Germany
Email: marc.bommert@hs-rm.de

***Abstract*—This paper presents an approach to segment parallelized algorithms in a mixed criticality multi-processor system. Segmentation is based on utilization of processors by higher criticality scheduling layers. The ambition is to establish load distributions with reduced overhead compared to non-clearvoyant distributions. The approach tends to be deterministic in order to guarantee real-time capabilities, i.e. bounded maximum execution time of parallelized segments, by avoiding thread migration completely. We present two methods of load segmentation: A static distribution based on WCET analysis of hard real-time tasks and a second method to overcome the issues of the static approach by means of backfilling.**

I. Introduction

A. Contribution

We present a novel approach to segment lower critical parallel algorithms in a partitioned mixed criticality (MC) multi-processor environment, based on information on the temporal behavior of higher critical tasks. It intents to reduce the overhead of workload distribution for the sake of throughput and to facilitate task parallelization, while preserving predictability to a maximum. The main target is to segment parallelized loops in a way such that the parallel subtasks achieve equal completion times, even if inhomogenous processing capacity remains available on each processor due to the partitioned execution of higher critical tasks.

Tasks of highest criticality are by definition well-analyzed in terms of their functional and temporal behavior to prove them correct. The result of system level temporal analysis is usually an execution time budget accounted to a task in order for it to meet its deadline in any possible system state. We show that the proportions between these time budgets can be consulted as an evaluation criterion for proper load distribution between lower criticality subtasks, resulting in less coordination effort, i.e potential earlier completion of the overall task. Since subtask/worker-thread migration is not required with this approach, it basically allows bounded worst-case execution times.

B. Organization of this Paper

In Section II, we describe the theoretical background. A suitable MC reference model is defined. The OpenMP standard for shared memory parallel programming is briefly introduced. A description of the identified problem scenario finishes Section II. In Section III, we present our approach of defining processor utilization functions and using these functions to control load distribution to tasks. We introduce two methods, a statically weighted distribution and a hybrid, partially dynamic distribution. We complete with simulation results and an outlook on further work.

II. Background

A. Conventions

To avoid ambiguity we define the terms used in this paper as follows: A *process* is an instance of a computer program. A process comprises one or more *threads*, flows of execution, which can be independently scheduled by the operating system on the system's processors. Threads providing means to execute arbitrary application-specific *tasks* as workload are called *worker threads*. A *real-time task* denotes a part of a real-time (RT) application with sporadic or periodic invocation and a certain *deadline*. We distinguish between *hard real-time* (HRT) tasks, which never miss their deadline, and *soft real-time* (SRT) tasks, which may miss their deadline, but guarantee bounded *tardiness* to still ensure a certain *quality of service*.

B. Mixed Criticality Scheduling

The MC concept has been introduced for the avionic domain in order to reduce size, weight and power (SWaP) of embedded systems. The fundamental idea is to lower the number of hardware platforms in a system by consolidating independent software modules onto the same platform. An important aspect of this reduction is proper fault isolation through *temporal* and *spatial partitioning*. Both, the SWaP paradigm and the MC concept, are promising for mobile applications in general, e.g. automotive ones, due to the rising importance of multiprocessor platforms and strong demand for fault isolation [1].

For transfering the MC approach from uniprocessor (UP) systems towards multiprocessor (MP) systems, since RT scheduling on an MP system is usually an NP-hard problem [2], a common approach is to partition the processors against each other for HRT tasksets. This reduces schedule verification efforts on MP systems to methods similar to those used on UP systems, which are well known.

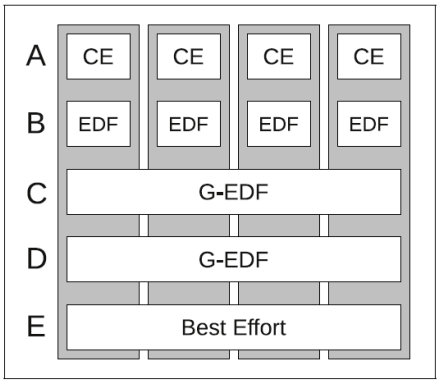


Fig. 1. Container scheduling in an MC environment, Mollison et al. [3].

Our reference model of an MC *stack* is inspired by MC^2 [3]. Figure 1 shows the MC^2 stack, with each of the nested layers implementing a specific scheduling algorithm:

Layer A uses a *cyclic executive* (CE) for applications of highest criticality with HRT constraints. The second-highest layer B uses *partitioned EDF scheduling* (P-EDF) for applications of highest

criticality with HRT constraints. A third tier, comprising layer C and D, uses *global EDF scheduling* (G-EDF) for applications with SRT constraints. Finally, the lowest layer E assigns remaining processing time with, for example, *global fixed-priority scheduling* to processes that are executed in a *best effort* manner.

We chose the mentioned scheduling stack for its hierarchical structure and for the degression of software criticality from highest towards lowest layers. Basically, a hierarchical scheduling model is sufficient for our purpose. However, in such a scheduling stack, tasks on each layer are preempted if tasks on a higher scheduling layer become eligible for execution. Preemption delays are accounted when RT behavior is analyzed. In practice, in contrast to the referenced MC stack, at layer B, rate monotonic scheduling (RMS) is probably more commonly used than P-EDF due to a simpler implementation based on priorities, which most RTOS provide, and, consequently, higher determinism in case of deadline misses. RMS, however, is not optimal for tasks whose periods are not harmonic. Thus, in most real-world scenarios, a certain processor capacity will remain unused by scheduling layer B. For non-periodic tasks in RMS, it is common practice to be mapped to periodically planned task-containers [4], with a period according to the highest possible activation rate. This additionally will lead to more processing time being accounted than being used in average when RMS applies at layer B.

C. Shared Memory Parallel Programming with OpenMP

Whilst using an MP platform allows parallel processing to execute independent single-threaded applications in parallel[1], it also allows algorithms to exploit real parallelism, e.g. performing segments of computations on multiple processors, collecting the results. Computational problems face a permanent rise in complexity and amount of to-be-processed data. Due to the challenge for modern UP systems to further increase circuit density and throughput to satisfy Moore's Law, this is one important method to cope with increasingly strong temporal requirements.

Parallel programming usually involves explicit definition of thread behavior in terms of synchronized execution and data access. Until today, it is a topic in research to simplify this process by abstracting to a parallel programming model which hides these issues from the more functional considerations of development. Such a standardized and widely used programming model is OpenMP [5]. It allows shared memory parallel programming within a single process, starting parallel worker-threads on demand and hiding their specific synchronization and data access. OpenMP imposes a thread *fork-join* structure to underlying software layers, i.e. the operating system (OS) scheduler.

Due to its size, its high level of abstraction and the methods of work allocation to be found in current implementations, e.g. distributing subtasks to processors from a global queue, OpenMP itself has not received much attention in development of RT systems. The fork-join model of OpenMP imposes tasks with *zero laxity* to the scheduler, which cause worse schedulability. Transforming fork-join task models to improve their schedulability is a topic in recent research [6].

An example of an OpenMP-parallelized loop is shown in Listing 1. A large number of iterations is implicitly parallelized into independent subtasks of equal weight by the preceding preprocessor statement. Threads are forked at loop entry and have to join at loop exit. There is no explicit concurrent data access. The shown example may implement any single-instruction, multiple-data algorithm with independent iterations, e.g. vector operations, tree or cube traversal, searching or even sorting.

[1]this can be legacy applications that are consolidated in an MC approach

Listing 1. C Language Example Code: OpenMP-parallelized for-loop

```
#pragma omp parallel for schedule (static)
for (int i = 0; i < 1000000; i++) {
  result[i] = processData(&data[i]);
}
```

D. Problem Scenario & Incentive

In a multi-processor MC system, highest scheduling layers work in a partitioned or clustered manner. Furthermore, application software of high criticality is verified with corresponding high detail regarding its temporal behavior. Worst-case execution time (WCET) is determined, accounted to tasks, and forms deadlines. For layers of lower criticality with SRT requirements or without temporal requirements, the partitioning is relieved and tasks are allocated to processors from a global ready queue.

Thus, software (with or without RT constraints) demanding parallel execution of algorithms, such as the fork-join model applied by OpenMP, is not directly considered by the MC approach. Due to inhomogenous remaining processing capacities on scheduling layers below partitioned HRT scheduling, throughput of parallel algorithms is not efficient. The distribution of parallel workload to processors is usually not aware of the demand for processing capacity at higher criticality layers, which is directly linked to parameters of real-time tasks at these layers. Due to excessive verification effort required at highest criticality layers, those tasks parameters are usually well-determined anyway. They could easily be used to optimize workload distribution, although it is important not to expose them to isolated tasks of lower criticality.

A trusted layer of workload distribution, e.g. an implementation of the OpenMP runtime libraries at lower levels of the MC stack, is suitable to overcome this issue in order to potentially improve efficiency while preserving fault isolation and preventing covert information channels from higher towards lower criticality processes. Furthermore, if a fixed mapping between worker threads and a system's processors would be used, RT guarantees could basically be given by means of bounded execution times of parallelized task segments.

III. A SCHEDULING LAYER FOR DIVISIBLE LOAD IN A MIXED CRITICALITY ENVIRONMENT

We define an additional scheduling layer in the presented MC model. This new scheduling layer is located below the two topmost partitioned scheduling layers A and B. Threads of that new layer are preempted when HRT-tasks of a higher layer become ready. In turn, activity in lower scheduling layers is preempted when either the new layer or HRT layers signal readyness. At the new layer, work is distributed globally for all processors of the system or a specified cluster of at least two processor nodes.

For simplification, we assume the new scheduling layer to contain a single process for now. This process interfaces an RT capable subset of the OpenMP programming standard. It implements a single parallelized loop as introduced in listing 1. When the loop is executed, a number of worker threads, one for each processor, become eligible for execution. Each of the threads is assigned a subtask to process a disjoint share of the parallelized loop. The *share*, i.e. the loop iteration count to be assigned to a specific subtask, is determined by a *distribution function* which itself decides about workload distribution based on a *utilization function*.

A. Utilization Functions

If a parallelized task is executed on a lower MC layer, below the partitioned HRT-tasks, a processor may be claimed by higher criticality tasks. Since the superior task schedule is static and a complete WCET analysis is available, information on worst-case processor utilization by HRT tasks can be statically provided as a utilization function $u(i, t_0, t_1)$. It delivers the worst-case utilization of processor i in $[t_0, t_1)$.

In a periodically planned partitioned MC system, a system *global period* T_p can always be derived as the lowest common multiple of all task periods of all partitioned, higher criticality tasksets. For a realtime system which measures and controls a real-world process with high frequency, task periods are usually short. Thus, the system global period is also potentially short. More precisely, our approach requires that the overall runtime of the parallel sequence is a large multiple of T_p. This allows to consolidate utilization functions for integer multiples of the system global period T_p with reasonable error, despite the exact execution time of the parallelized task segment.

T_p is the period of the utilization function. The function result is a normalized value: A result of 0 indicates an idle processor i and a result of 1 indicates that the processor will not be available at all in the given interval. The utilization function is defined at system integration time, based on static HRT task parameters, e.g. their period and their WCET.

For the following definition of workload distribution methods, we distinguish between worst-case utilization of each processor, based on WCET analysis of RT tasks, and its best-case utilization. Therefore, we distinguish between worst-case and best-case utilization functions, $u_{WCET}()$ and $u_{BCET}()$.

B. Weighted Static Distribution of Parallel Load

From $u_{WCET}()$, certainly remaining processing time of processor i can be determined as $1-u_{WCET}(i, t_0, t_1)$. The proportions between worst-case processor utilization define a proportional distribution function $z(i, N, m, t)$ which is consulted at each fork operation for each of the involved processors $[1..m]$:

$$z(i, N, m) = \frac{1-u_{WCET}(i,0,T_p)}{\sum_{k=1}^{m}(1-u_{WCET}(k,0,T_p))} * N = N_i$$

Where N is the overall loop iteration count as illustrated in listing 1, and N_i is the loop iteration count to be distributed to processor i.

This proportional distribution function z is consulted during execution of the fork operation of the parallel task segment. It is called exactly once per processor per parallelized loop to decide about the proportions in which N is shared between subtasks. Load segmentation according to z would be optimal, if the effective execution time of higher criticality tasks would always equal their WCET and the runtime of the parallel task segment would be an integer multiple of T_p. Depending on the execution time variance of higher criticality tasks, and the parallel segment's exact execution time, the distribution function z is thus presumably always affected by a certain error. This error has to be tolerated to give RT guarantees to the parallel task segment. Dividing the parallel segment based on WCET analysis of higher criticality tasks will thus barely ever achieve an optimal result. Nevertheless, this *static weighted* approach is able to reach better results than a static distribution not considering processor utilization by higher criticality tasks, which we will further refer to as *static naive* for comparison.

Using the static weighted method, *starvation* of parallel subtasks is prevented by design. It is technically easy to limit the maximum processor usage and to inherit definitive processing time to lower MC layers.

C. Hybrid Distribution of Parallel Load

In order to overcome an issue of a static weighted distribution based solely on the WCET of higher criticality tasks, a *hybrid distribution* concept seems promising. Instead of entire static segmentation of load, a hybrid algorithm would pre-divide load into a fraction N_s which is assigned statically and a second fraction N_d, which is assigned dynamically. This allows threads finishing their statically assigned share early, to actively claim new work. Such *backfilling* is potentially capable to reduce or eliminate inserted idle time caused by reduced utilization of a specific processor at higher criticality layers. Its cost is an increased number of potential synchronization operations between workers when dynamically requesting new work, i.e. a certain fixed number of loop iterations.

To pre-segment the overall number of loop iterations, the band between WCET and BCET in each processor's RT-taskset may be considered by means of the utilization functions u_{WCET} and u_{BCET}. The dynamically allocated part has to be sufficiently large to compensate the expected imbalance in processor utilization due to execution time variances of higher criticality tasks. Therefore, processing capacity definetly not used by a processor by HRT tasks $c_{def} = 1 - u_{WCET}$ is distinguished from utilization that may additionally not be required $c_{pot} = u_{WCET} - u_{BCET}$. N_s and N_d can be determined to:

$$N_s = N - N_d = N * \frac{\sum_{i=1}^{m} c_{def}}{\sum_{i=1}^{m}(c_{def} + c_{pot})}$$

By choosing the number of operations which are dynamically allocated on worker thread request, speaking in OpenMP terminology, the *chunk size* of dynamic work allocation, this approach trades between worker synchronization overhead and the finally remaining maximum inserted idle time. The divisibility factor, i.e. the WCET of a single loop iteration, provides an ultimate limit.

IV. Simulation Results

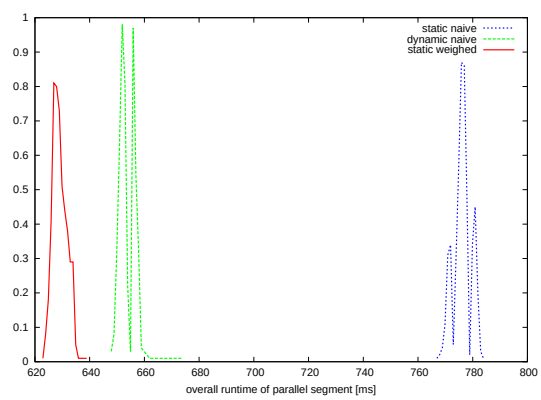


Fig. 2. Static weighted distribution with constant load from a higher priority task.

Figure 2 shows simulation results comparing three types of workload distribution methods. The simulation platform is a dual-core processor running a vanilla Linux kernel. One of the two

processors is constantly kept under load to a known share ($\frac{1}{3}$) by a highest priority thread simulating a cyclically executed RT task with a period of 3ms. The shown plot compares execution time of a parallel task segment subject to a static naive workload distribution, a fully dynamic workload distribution, and our (clearvoyant) static weighted distribution which allocates shares according to remaining processing capacities. A single work cycle iteratively computes a short Fibonacci sequence. The workload consists of 100 million cycles. Dynamically assigned chunks consist of 1000 single cycles. Each simulation is repeated 500 times.

It can easily be seen, that the static weighted distribution outperforms the naive distributions. Its main advantage compared to the static naive distribution is the drastically reduced inserted idle time at the processor node which is idle at the highest criticality layer. The advance to the fully dynamic (naive) distribution is due to reduced worker synchronization.

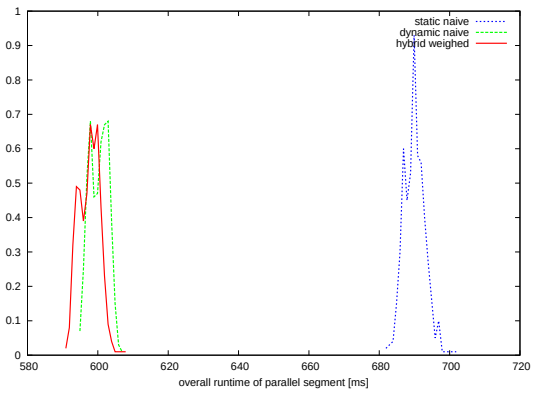


Fig. 3. Hybrid distribution with variable load from a higher priority task.

Figure 3 shows simulation results comparing three types of workload distribution methods. One of the two processors is dynamically kept under load within known bounds ($\frac{1}{6}$ to $\frac{1}{3}$) by a highest priority thread simulating a cyclic executed RT task with a period of 3ms. The shown plot compares execution time of a parallel task segment subject to a static naive workload distribution, a fully dynamic workload distribution, and our (clearvoyant) hybrid distribution. Simulation parameters are as mentioned before, but dynamically assigned chunks consist of 1 million single cycles.

Again, we observe a clear advantage of our distribution over the static naive distribution. More interesting, the hybrid method also performs slightly better than the dynamic naive distribution, although the latter is subject to only 100 work-stealing cycles, i.e. worker thread synchronizations, during each run.

V. Conclusion

A. Discussion

In this paper, we described a novel approach for scheduling divisible load in an MC system at a layer which is subordinate to partitioned HRT scheduling. We proposed a method to consider parameters of RT tasks in order to determine an adequate segmentation of large numbers of independent calculations. We focused on calculations with equal demands of processing time, e.g. OpenMP-parallelized for-loops.

We concluded that the RT system could provide utilization functions to predict definetly and potentially unused processing time for each processor. Data provided by such functions, based on static information from the system integration domain, could be used to segment divisible load regarding the optimality principle, i.e. reaching equal completion time of subtasks. The resulting optimization towards the optimality principle would improve efficiency of parallel task execution.

Since subtask/worker-thread migration is not required with this approach, it basically allows bounded worst-case execution times and, consequently, provides a HRT-capable interface for parallelized algorithms.

B. Future Work

We currently develop a prototype implementation of the presented approach by gradually establishing a subset of an OpenMP-compatible environment on basis of an RTOS. This environment will have to interface the integration domain to gather highest criticality task parameters, i.e. WCETs and BCETs. Load distribution within this environment will then consider these parameters. Prior attention lies on parallelized loops, that is, dynamic (runtime) assignment of loop iterations to worker threads. So far, we can say that the scheduling methods which can be interfaced through the OpenMP API, namely *static, dynamic* and *guided*, suit the presented approach. Such a middleware layer aims towards a general purpose solution. Static assignment (pinning) of worker threads to processors, disallowing the (non-clearvoyant) OS scheduler to migrate them between processors, will presumably allow execution time guarantees to be given in order to fulfill HRT requirements. Verification has to consider specific, yet to be defined, use cases reproducing real-life scenarios such as processing multimedia data with and without RT constraints.

In future work, the method can then eventually be extended towards multiple parallel task segments, nested parallel task segments and parallel task segments that explicitly access shared data and thus are subject to locking. The approach may also be adapted to parallel segments with inhomogenous runtime which itself are weighted against each other by means of annotations. Limits in memory- and bus-bandwith have not been considered yet, but are crucial in a real-world implementation and thus need further attention. Methods towards the system integration domain have to be designed in order to automatically define utilization functions based on given or gathered HRT task parameters.

References

[1] ISO 26262, "Road Vehicles Functional Safety," 2011.

[2] K. Ramamritham, J. Stankovic, and P. Shiah, "Efficient scheduling algorithms for real-time multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 01, no. 2, pp. 184–194, 1990.

[3] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos, "Mixed-criticality real-time scheduling for multicore systems," in *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, ser. CIT '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1864–1871.

[4] L. Sha, M. Klein, and J. Goodenough, "Rate monotonic analysis for real-time systems," Carnegie Mellon University, Software Engineering Institute, Tech. Rep., 1991.

[5] OpenMP Architecture Review Board, "OpenMP application program interface version 3.0," May 2008.

[6] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*. IEEE, 2010, pp. 259–268.

Application Architecture Adequacy through an FFT case study

Emilien Kofman [1,2] Jean-Vivien Millo [1] Robert de Simone [1]

[1]INRIA Sophia-Antipolis, Aoste team (INRIA/I3S/CNRS/UNS), 06560, Sophia-Antipolis, France
[2]Univ. Nice Sophia Antipolis, CNRS, LEAT, UMR 7248, 06900 Sophia-Antipolis, France
{emilien.kofman, jean-vivien.millo, robert.de_simone}@inria.fr

ABSTRACT

Application Architecture Adequacy (AAA) aims at tuning an application to a given hardware architecture. However it is still a difficult and error prone activity. As like as in Hardware/Software co-design, it requires a model of both the application and the architecture. With the new highly-parallel architectures, AAA should also allow a fast exploration of different software mapping granularity in order to leverage better the hardware resources without sacrificing too much productivity. The main contribution of this paper is to extract from a case study a methodology based on dataflow modeling to make the software both faster to develop and suited to the target. Then we show how this methodology can solve some of these issues.

Keywords

Application Architecture Adequacy (AAA), Fast Fourier Transform, Hardware/Software co-design, Massively Parallel Processor Array (MPPA), parallel computing

1. INTRODUCTION

This article identifies through a case study how to make the maximum use of the heterogeneous parallelism of the upcoming architectures. The example we picked is the well known FFT algorithm. It is often taken as a benchmark utility but one should keep in mind that this is almost always a building block of larger software systems, and not a standalone application. Thus it is important to study its potential parallelism when running within the regular conditions. Moreover, the chip-level parallelism has a growing interest because it allows a large and scalable parallelism. It is now possible to aggregate many cores on the same chip but at some point the bus medium for data transfers becomes the bottelneck because communication is serialized, hence the need for new communication media. This problem can be solved using network on chips (NoC) which allow parallel communications. However it also raises much more complexity for the programmer.

It is possible to adapt an implementation to a given architecture although this work often requires many modifications and is thus very time consuming and error prone. On contrary, given an implementation it is difficult to pick an architecture which would improve it (either for performance, energy consumption, temperature, cost). The reasons are that there exist only few tools for fast architecture design space exploration (such as [4, 7] based on systemC) and they often require expertise. Moreover the implementation sometimes already assumes hardware specific features thus the need for a new representation.

2. CASE STUDY

In order to identify which representation would fit we picked a DSP algorithm and tried to adapt it to a given system, while keeping in mind which choices are related to specificities of the given hardware system.

2.1 Hardware architecture

Very different, and heterogeneous hardware architectures exist. General purpose GPU offer a very massive fine-grain parallelism while multi-core CPUs offer coarse-grain parallelism. Many-core architectures fall in the middle. The experimental platform is the Kalray Massively Parallel Processor Array (MPPA-256). It has 16 clusters of 16 VLIW cores, which yields a total of 256 VLIW cores. The clusters are connected through a network-on-chip which is accessible through a message passing interface. The cores inside a cluster are connected through a bus and share a 2MB memory, then the parallelism is leveraged thanks to openMP. Additional cores are available for input/output purposes (PCIe/Ethernet/GPIO/Interconnect). Thus one Kalray MPPA-256 machine offers three levels of parallelism:

- The compiler bundles instructions for VLIW cores and thus provides an instruction level parallelism.
- A shared-memory intra-cluster parallelism using openMP or POSIX threads.
- A message-passing inter-cluster parallelism using a specific message passing interface.

2.2 FFT implementation

A from-scratch iterative Cooley-Tukey decimation in frequency implementation allows to understand better the data dependencies in this algorithm. Decimation in frequency was preferred instead of decimation in time because it splits the dataset in half at each stage instead of splitting even/odd sample indices. It is thus easier to experiment with, especially on a distributed memory architecture. However, a DMA-assisted transfer could efficiently split even/odd samples.

The FFT implementation often comes with different steps (normalizing, FFT, unscrambling). The FFT *Step* requires multiple stages which are a set of mutiply-add operations named *radix*. For instance, a 2^{13} samples *radix2* FFT has 13 stages of 2^{12} *radix2* operations. In this paper, *Step* is disambiguated from *Stage* which is part of the FFT *Step*.

The radix operation is the building block of the FFT algorithms. Optimized routines exist for *radix2* to *radix16* (including *radix3*, *radix5*,...). It is sometimes called a butterfly operation due to its datapath representation. It is made of multiply-add operations and requires constant coefficients named twiddle factors. When done in-place (one buffer for both input and output), the FFT algorithm outputs results in bit-reversed order. Thus, the samples need to be sorted, this is the unscrambling step. The resulting samples often need a normalization factor which can be applied either at the end or along the FFT stages. In order to check the implementation performance, the pseudo-throughput is defined: let N be the number of samples:

$$throughput(Gflops) = \frac{5.N.log_2(N)}{time(ns)} \quad (1)$$

An inplace transform is implemented and the twiddle factors are pre-computed. The bit-reversing steps and normalizing steps were implemented for functional testing but are not taken into account in this study because depending on the whole application, they may not be necessary. Moreover the bit-reversing is sometimes hardware-accelerated (some DSP are capable of bit-reversed addressing). The algorithm shows that at each stage of the FFT, all the radix operations could run in parallel (provided sufficient computation units). Then, synchronization is needed at each stage and the stages could be pipelined (provided sufficient memory). This is the ideal, maximum parallelism of the application which is reached for example in hardware implementations or with GP-GPU implementations [9].

One should keep in mind that most of the signal processing applications will use the FFT on a dataset with a power-of-two number of elements, and within a given range (usually not larger than 2^{12}). Moreover, applications will very-likely run batches of FFT, and not a single one (for instance for image processing purposes).

Apart from GPU implementations, few work exist on the highly parallel implementation efficiency of FFT on distributed memory architectures, and they either conclude that the sequential implementation runs faster ([3]) or that the parallel implementation runs faster for a very large (non-realistic) dataset of more than 2^{12} samples. Other parallel implementations study only multidimensional FFTs, which fall in the scope of "batched FFTs", and thus gives better results than one-dimensional FFT on reasonable datasets because they allow a simpler data-parallelism. For instance when running a 2D-FFT, one can run first 1D-FFT on each row (and they are independant), then run 1D-FFT on each column.

Although this study focuses on parallel implementations, sequential efficiency of the algorithm is of course important. *radix2*, *radix4* and *radix8* versions of the algorithm have been implemented and *radix4* and *radix8* clearly outperforms *radix2* (by a factor of 3.3x on x86 CPU). Mixed-radix has not been implemented.

2.3 Results

The implementation first focuses on shared memory parallelism achieved with openMP (Figure 1 on x86 and Figure 2 on MPPA), then evaluates distributed-memory parallelism. The provided tools allow to compile both for Kalray's architecture and for the Host's architecture (which is an Intel i7-3820 CPU with 8 cores). The scale on the right gives the pseudo throughput (equation 1). White edges identify the number of threads which allows highest throughput for given FFT size. The sequential column shows performance when openMP pragmas are ignored, which differs from the 1-thread column (openMP pragma are not ignored but the number of threads is restricted to 1).

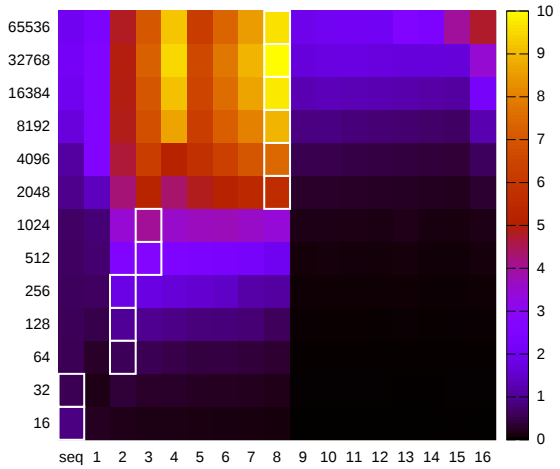


Figure 1: FFT size over number of threads on host

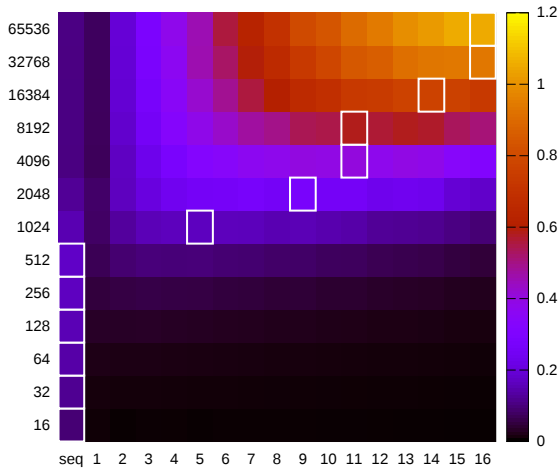


Figure 2: Evaluation on one 16-cores cluster

As you can see, there is significant speedup in both cases but there are also few differences. The absolute performance of the CPU is higher than on one MPPA cluster. The most interesting result is probably that the parallel implementation outperforms the sequential implementation even for small sizes on CPU, but only for "large" sizes (more than 2^{10} samples) on one cluster. This may be due either to different openMP implementations or to the fact that VLIW cores already perfom the radix steps in fewer operations, thus reducing the computation time over sync time factor. Figure 3 gives the maximum speedup performance results. This shows a 11x speedup for large FFT sizes. Although we admitted that such large datasets are not often used, this

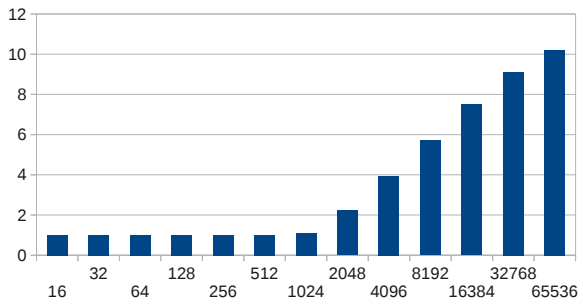


Figure 3: Maximum speedup on one MPPA cluster

gives a clear idea of the reachable speedup performance for batched FFTs or multidimensional FFTs because it is possible to run only a subset of the 2^{16} FFT stages such that it computes multidimensional FFTs of smaller sizes (reciprocally, a large FFT can be computed combining the results from smaller FFTs). Given these results, it is probably wiser to run batches of FFTs to leverage best this architecture. This means this architecture is not very well suited to fine grain parallelism but can achieve massive coarse grain parallelism. The figures shows that provided low communication overhead it would probably outperform CPU when running batches of FFTs on 16 clusters.

This experiment rises two problems: The application and the architecture needs to be described in such a way that it is easy to allocate resources differently given the same (validated) implementation. The description of the architecture needs to be precise enough in order to formally decide the best suited parallelism granularity for a given hardware architecture.

3. MODELING METHODOLOGY

Deciding which amount of parallelism should be automated is now an actively studied topic. The related works in this area yield at least two main methodologies.

Some methods would take legacy code and compile it through a front-end to an intermediate (possibly hardware-independent) representation [2]. Then identify parallelism in this representation with a custom tool and apply back-end transformations according to the given architecture. This source to source compiling allows to identify fine-grained parallelism (e.g Instruction Level Parallelism) which is for instance necessary when compiling code for a VLIW architecture but extracting coarse grain and pipeline parallelism is not easy.

Another way to tackle this problem is to express applications in a dataflow-representation (either with a text-based or graphical language) in order to ease the identification of parallelism, then compile it to a given architecture. Given the appearence of specific languages for specific architectures (e.g. Open Computing Language for GPUs) it is reasonable to think that a new representation is needed for signal processing and multimedia applications.

3.1 Motivations

Many attempts exist in this area ([1, 5]) but for instance in the case of StreamIt[5], the description of the architecture is limited to sparse information (number of threads, size of caches), and no model of the architecture is provided in order to help the allocation mechanism. This can result in under-performing implementations. For instance using sockets for inter-process communication, either when processes are located on the same machine or on another machine on the network provides homogeneity to the whole compilation process, but using the shared memory would be more efficient. Thus compiling a streamIt application for another architecture involves changing the compiler's behavior which is a tough task. GUI-programming tools also exist when it comes to mapping dataflow applications onto hardware. They allow very comfortable learning and fast prototyping but they do not compete with hand-written applications. Moreover they are sometimes bound to specific hardware.

3.2 Dataflow graph representation

Synchronous Data Flow ([6], SDF) is a dataflow process network used to express logical parallelism of data flow applications. A functionnally correct representation of an application within SDF allows formal checking for deadlock, starvation, conflict. Moreover it is now admitted that it eases analysis of the buffer size over throughput compromise and thus allows further optimisation for instance through static scheduling [8].

An SDF is a graph structure in which every vertex has a type. The graph has a set of agents N, a set of places P and a set of arcs. The edges of an SDF are directed, they are hence called *arcs*. An arc cannot connect two vertices of the same type. An arc in an SDF has a width expressed with a non-zero integer that represents the number of tokens travelling simultaneously on it. The places hold tokens. Each place has exactly one incoming and one outgoing arc.

This representation makes no assumption on the architecture (buffer sizes, execution speed). In the scope of this article, an SDF models an application where the agents represent the different filters (or actors) that can be performed concurrently in the application. The places represent a location in memory. The arcs represent the flows of data (data dependencies). The presence of a token in a place represents the availability of a data element in the memory. An agent without incoming (outgoing) arc represents a global input (resp. output) of the application. The arcs does not necessarily describe an access in memory or a channel of communication but a flow of data between agents. The nature of the link will come with the description of the architecture.

It is then possible to give a (very fine grain) SDF graph of the FFT algorithm which is actually very close to the well known butterfly diagrams, which exposes the maximum parallelism. Expressing the same algorithm serially obfuscates the data dependencies. Because of the very repetitive patterns it would be easier to represent the application with a language or a set of classes (as like as streamIt or FastFlow) and not graphically.

3.3 Morphing and mapping

Given a precise description of the architecture, this SDF representation can be morphed to the correct parallelism level, then mapped to an hardware architecture. For instance if the FFT has to be implemented on GPU, the very fine grain representation could be fine. However if it has to be implemented on CPU, it would be very time consuming to synchronize that much threads (assuming one agent is

mapped to one thread) thus it is not the correct representation: on contrary the Figure 1 shows that it is essential to limit the number of threads accordingly to the number of cores.

Indeed, a precise description of the architecture and its communication media should allow to split and merge agents depending on communication throughput/latency, DMA engines, routing in the case of network on chips and depending on the computation elements (size of their cache and local memories, co-processors, VLIW or SIMD features, ...). Then, the actual data moves can easily be identified. We introduce the following morphed and mapped FFT representation (Figure 4). The assumed hardware in this example is an heterogeneous shared/distributed memory architecture as like as the Kalray MPPA-256. Two places which are in the same cluster can benefit from shared memory (reduces communication time and memory consumption compared to a FIFO). Only the edges from one cluster to a different cluster require message passing. These clusters have a DMA thus the message passing could be asynchronous.

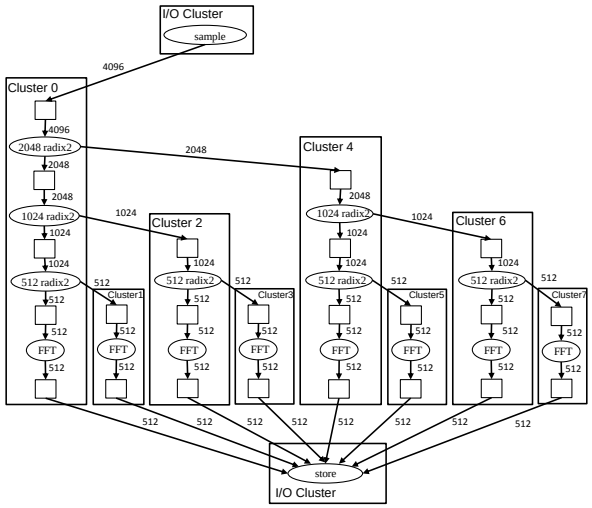


Figure 4: A 4096-samples FFT algorithm represented with a dataflow, then morphed to fit an architecture composed of 8 distributed memory clusters.

Further investigation on a 4096 samples FFT shows that assuming no shared memory parallelism, positive speedup is obtained when splitting the first stage, but not when splitting the second stage. However combining openMP and message passing in this manner does not provide positive speedup.

4. FUTURE WORKS

The tough task of the morphing and mapping steps is to decide the representation level to allow a correct exploration for a performant implementation. Indeed the architecture has to be described but the abstraction level is not yet identified. Assuming the heterogeneities (for instance between a regular CPU, a processor array, and a GPU), it is clear that at least a high level representation (UML/SysML) of the hardware is needed and not only sparse information. Different level of complexity exist: non-functionnal UML, systemC-TLM or CABA (Cycle Accurate Bit Accurate), ISS, complete IP-XACT descriptions. An other obstacle to this exploration is that it is sometimes hard to obtain precise information about the hardware, especially for specialized chips (GPUs, accelerators).

5. CONCLUSIONS

The present paper explained through an example how software development of DSP and multimedia algorithms could improve in order to ease code reuse and validation on mutiple hardware targets. The dataflow representation, and especially the SDF representation of applications [8] comes naturally as a suitable candidate for hardware-independent and optimisation capable representation. However picking a hardware representation for fast design space exploration and for performant implementation is still complex.

6. REFERENCES

[1] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: high-level and efficient streaming on multi-core. In S. Pllana and F. Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, Jan. 2013.

[2] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. O. McMahon, F.-X. Pasquier, G. Péan, P. Villalon, et al. Par4all: From convex array regions to heterogeneous computing. In *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, 2012.

[3] M. Balducci, A. Choudary, and J. Hamaker. Comparative analysis of fft algorithms in sequential and parallel form. In *Mississippi State University Conference on Digital Signal Processing*, pages 5–16, 1996.

[4] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri. Mparm: Exploring the multi-processor soc design space with systemc. *Journal of VLSI signal processing systems for signal, image and video technology*, 41(2):169–182, 2005.

[5] M. I. Gordon. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. PhD thesis, Massachusetts Institute of Technology, 2010.

[6] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceeding of the IEEE*, 75(9):1235–1245, 1987.

[7] LIP6. The soclib project : An integrated system-on-chip modelling and simulation platform. http://www.soclib.fr/, 2003.

[8] J.-V. Millo and R. De Simone. Periodic scheduling of marked graphs using balanced binary words. *Theoretical Computer Science*, 2012.

[9] V. Volkov and B. Kazian. Fitting fft onto the g80 architecture. *University of California, Berkeley*, 40, 2008.

Running Linux and AUTOSAR side by side*

Tillmann Nett, Jörn Schneider
Trier University of Applied Sciences
Schneidershof
Trier, Germany
{T.Nett, J.Schneider}@Hochschule-Trier.de

ABSTRACT

Mixed criticality systems are systems, on which safety-critical and non safety-critical software must run simultaneously. For such a system it is necessary that all deadlines of safety critical jobs can be met, and no safety-critical function is impaired by any other function. Current approaches for designing such a system include virtualization, hardware partitioning or implementing all software as critical software. These approaches however introduce additional costs due to additional hardware, more complicated development techniques for non-critical software or loss of processing power due to the virtualization layer.

We demonstrate a novel method for implementing a system, that provides lean interfaces for real-time software and a full Unix interface for non real-time software. This system uses vertical partitioning to run two different operating systems on two cores of a single ARM multiprocessor.

1. INTRODUCTION

Real time operating systems such as AUTOSAR [1] allow provably fast responses to physical events. This is required for example in cars, where the speed of the wheels has to be continuously monitored to take action if the brakes lock. As a failure of such systems often can harm lives, such operating systems need to be simple and easy to verify for correct functional and temporal behaviour.

Interactive operating systems provide a good average case performance thus delivering fast responses to user request in most cases. However, they are not designed for predictability and the worst-case reaction times can be very high.

In a growing class of systems users interaction, e. g. through a GUI (Graphical User Interface), and guaranteed real-time behavior of safety relevant functions has to be available simultaneously. Such systems can be considered as a kind of mixed-criticality systems. This means that such a system must at the same time fulfill requirements for two types of systems, which cannot easily be provided by a single operating system.

A viable but impractical method for building such a hybrid system is to implement all parts to the same standards according to the highest level of criticality on top of a common real-time operating system. This means all interactive parts, including driver software and the complete user interface would have to be implemented as real-time tasks. This introduces additional costs during development and whenever parts of the system are changed.

In some cases it may also be possible to change an existing interactive operating system to provide real-time facilities. For Linux various patches add additional schedulers, synchronization primitives etc. [11]. However the changes to the Linux kernel are often quite large as for example in case of the PREEMPT_RT patch. Also it is often unclear whether these patches offer full hard real-time guarantees or only soft real-time guarantees. Furthermore all kernel code would have to satisfy the relevant safety standards such as [10] and be included in worst-case execution time analyses.

Another method to build such systems is to put different parts of the system on different hardware each with a single operating system, and then connecting them via a bus or network. This method however requires doubling the amount of hardware build into the automobile and introduces latency because of the communication via the network. Furthermore this method increases the power consumption of the system.

A third design method is to add a separate partitioning layer underneath the real-time and the interactive layer following the principle of [4]. This layer separates the real-time system from the interactive system by assigning individual slots in a major time cycle and separating memory as well as forcing all communication over a so-called system partition. However, the enforced indirection has several disadvantages as described in [12].

We present a novel method for designing a mixed criticality system, running two different operating systems (vertical partitioning) on two Cortex-A9 cores. A similar method has previously been used to run two differently configured Linux kernels on two separate cores of an x86 System [8]. Other vertical partitioning setups first start the Linux system and then set aside a separate core for real-time work using core isolation methods provided by Linux [5]. This however greatly increases the startup time of the real time portions as the complete Linux system has to be up and running before the real-time core can be isolated.

The implementation method described here was used for

*This work is partly funded by the German Federal Ministry of Economics and Technology under grant number 01ME12043 within the econnect Germany project.

implementing an interactive monitoring and control system in electric cars for a field trial on user acceptance of smart grid technology in the project econnect-Trier.

The rest of this paper is organized as follows: Section 2 shortly summarizes the requirements, which governed our design and implementation process. In section 3 we give a detailed overview of the implementation of our prototype, including startup of both operating systems, and the static resource sharing scheme. In section 4 we summarize our work and show how future work can extend our implementation.

2. REQUIREMENTS

The main driving force for this research were the conflicting requirements presented in Section 1. However at the same time other non-functional requirements also had to be taken into account. We will now present the detailed requirements and their justifications.

1. It shall be possible to derive and prove hard real-time bounds for tasks where needed.
2. It shall be possible to implement non real-time or soft real-time parts using operating system abstractions well known by desktop programmers.
3. It shall be easily possible to argue and verify that the real-time tasks are not impaired by the non real-time functions.
4. It shall be possible to add, change or remove non safety-related software without additional analysis or verification of the real-time software.
5. It shall be possible to derive a fixed deadline for the boot time, i.e. the real-time parts of the system must be responsive after a fixed time when the system is started.

Requirements 1, 2, 3, and 4 have been justified in Section 1.

Requirement 5 has been defined, as in cyber physical systems often rigorous constraints are imposed considering answer time of systems. For example safety critical systems in cars often have to answer to messages on the CAN-bus within 100ms after system startup. This often requires large optimizations of the boot code when implementing such devices [2]. Our method on the other hand will optimize the response latency after boot by first starting the real-time operating systems and then starting the non real-time operating system on a separate core.

3. SYSTEM CONFIGURATION

In a uniform memory architecture two or more separate processor cores operate on the same shared memory. All cores can access the same hardware and can receive the same interrupts. For the mixed-criticality system a OMAP4460 multiprocessor [9] was used, which provides two cortex-a9 cores [6], two cortex-m3 cores as well as other specialized cores. Only the two cortex-a9 cores were used and all other cores were deactivated, to keep the system uniform.

3.1 Startup Sequence

During startup of a OMAP4460 the Cortex-A9 *Core0* is initialized first by the ROM code. The ROM code fetches the boot code from non volatile storage and starts executing it. At this point, the system runs in single core mode and can perform initialization steps and load the operating system. At the same time the *Core1* is put into an idle state by the ROM code, from which it can be awakened by the operating system. Once the operating system is ready it can configure the start address of *Core1* and send a special signal which awakes the other core. At this point the system runs in dual core mode.

Hardware	Assigned to
Screen	Linux
Touch-pad	Linux
non-volatile memory (SD-Card)	Linux
UMTS Module	Linux
USB Subsystem	Linux
L3 OCM_RAM (SRAM)	AUTOSAR
GPS-Module	AUTOSAR
CAN-Module	AUTOSAR
SPI-Interface	AUTOSAR
UART-Interface	AUTOSAR
L3 and L4 Interconnects	AUTOSAR/Linux
Main Memory (DRAM)	AUTOSAR/Linux
Interrupt Distributor	AUTOSAR/Linux
Interrupt CPU Interfaces	One per Operating System
Hardware Spinlocks	AUTOSAR/Linux

Table 1: Hardware distribution among operating Systems

The mixed criticality system uses *Das U-Boot* [7] as a bootloader to load an AUTOSAR conforming operating system from non-volatile storage. This AUTOSAR image currently also includes a complete linux kernel statically linked in a separate section of the executable. The real-time system used is an open source version of the ArcticCore AUTOSAR implementation [3]. This Version was implemented for a single core machines, hence the second core is not started by default. We added additional startup code which configures and starts the other core. On the second core a short startup routine is used to load the Linux image contained in the AUTOSAR image and transfer control to the Linux image. All startup parameters needed by linux are provided by the core1 startup routine and are written to the correct locations in memory. These parameters also include the `maxcpus=1` option, which instructs the Linux kernel to run in single core mode.

Das u-boot currently uses different code, depending on the operating systems to be started. For Linux this code also deactivates all interrupts, flushes caches and turns of caches and the MMU. This code is not performed when starting AUTOSAR from a ELF-image, but is required to later start Linux from within AUTOSAR. To put the system into a state that allows Linux to be booted, a call to this code was manually added to the ELF-startup code of das u-boot. As ArcticCore does not use caches or the MMU, it was not impaired by these configuration changes. Interrupts are later initialized again by code which was added to ArcticCore in a way that interrupt lines could be statically assigned to one of the Operating systems.

3.2 Hardware Assignment

To simplify the design of the system and to avoid the need for resource management protocols, which may impair real-time functions, we decided to statically assign most hardware to one of the two operating system. However some Hardware was needed by both operating Systems. For this hardware is is necessary that initializations done by

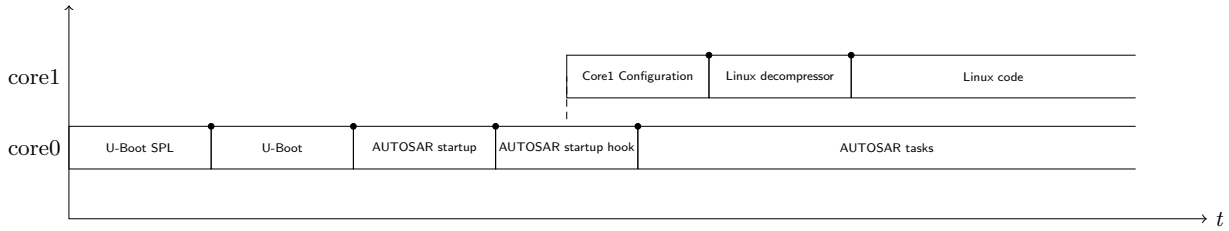


Figure 1: Startup sequence of the mixed system

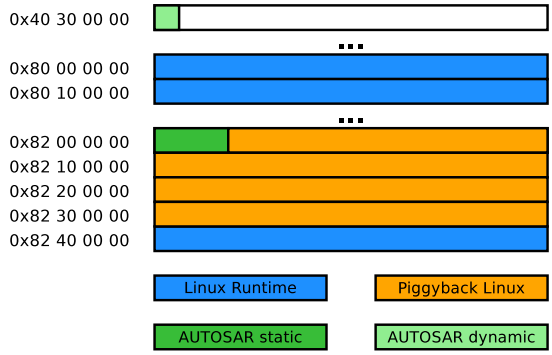


Figure 2: Memory configuration of the final mixed system

AUTOSAR are not be overwritten during booting of Linux. Hence for all shared hardware which is normally initialized by Linux, the initialization routines were deactivated and if necessary the initialization was performed during startup of AUTOSAR. One exception to this rule was the main memory. Main memory is initialized even prior to the start of the OS in das u-boot, hence this initialization could be kept in place. However to ensure that the address space reserved for AUTOSAR would not be used by Linux, we manually deactivated this address space in the Linux kernel using the `mem=` boot parameters. Like all other boot parameters, these were dynamically written by the *Core1* startup code within AUTOSAR, based on the actual address space used by AUTOSAR.

To ensure timing correctness for memory access the MMU was disabled on the core running AUTOSAR. For OMAP4460 SOCs disabling the MMU also disables all caches for memory used by that core. To provide the real time parts with a fast memory the L3 OCM_RAM (SRAM) was used for data parts of the system. The L3 and L4 Interconnects used for communication between subsystems as well as memory of the OMAP4460 was shared between subsystems. A closer analysis of the arbitration strategies for the Interconnects to ensure that all deadlines can be met is still required.

3.3 Interrupt Distribution

Both Cortex-A9 Cores in the OMAP4460 share a single generic interrupt controller (GIC) [1]. This GIC is responsible for global masking of specific interrupts and distributing interrupts to the cores. The GIC is divided into a global distributor and one CPU-interface per core. The distributor can distribute interrupts to one or more CPU-interfaces which then signal the interrupt to the CPU. Interrupts can then be acknowledged on the CPU-Interface. Interrupts can both be masked in the CPU-interface by setting a flag, as well as in the distributor by setting an empty target list.

Neither the application context of the described system, nor the chosen assignment of hardware resources require to signal interrupts to more than one core. At startup all interrupts are configured by AUTOSAR with an empty target list. The target is then set to the appropriate core when activating the interrupt in AUTOSAR or Linux. Reconfiguration of the target within the distributor requires setting a single bit in a memory mapped register. As other bits in the same register may be owned by another operating system, a lock had to be added around any configuration code. For this Lock one of the hardware spinlocks provided by the OMAP4460 SOC was used. Hardware spinlocks use memory mapped registers for operation, hence providing a fast way for synchronization between cores. This lock is only taken during short sections within the Linux kernel. Before requesting this lock we disable preemption in Linux. Because preemption is disabled, only a single kernel-thread can wait for the lock at any time. This means, the time that any AUTOSAR task must wait for the lock is bounded and a worst-case execution time analysis for AUTOSAR tasks remains possible, by taking this additional lock contention time into account. In case Linux is run on multiple cores however the lock could be requested by multiple kernel threads simultaneously. In this case it is possible for one of the threads to steal the lock from the waiting AUTOSAR task and the locking time becomes unbound. In this case a different synchronization primitive is necessary to ensure bounded execution times within AUTOSAR. For example a mixed-criticality lock [12] could be used. Within AUTOSAR all interrupts are configured during startup of the system, so that lock contention times only influence the startup and schedulability is not impacted by the lock.

3.4 Communication

To share data between AUTOSAR and Linux we added real-time communication facilities. For stream transmissions a non-blocking ring buffer implementation is used. Multiple ring buffers are provided for multiple data streams. Using these ring-buffers AUTOSAR tasks can send data packets to Linux, which are then received in a kernel thread. Multiple tasks sending data simultaneously through the same ring buffer must be synchronized with each other. For this the real-time resource sharing mechanisms within AUTOSAR are used.

For communication from Linux to AUTOSAR only the transmission of single word signals was needed. Hence these words were reserved within a shared memory space. These memory words are written in Linux using an atomic store operation and read within AUTOSAR using an atomic read

operation.

All data structures needed for communication are set up by AUTOSAR within the address space used by AUTOSAR. The addresses of these data structures are then transmitted to Linux prior to startup using a boot parameter. These data structures are then read from a Linux kernel module which configures all kernel data structures and offers a device file for communication with user space. The provided communication facilities match the requirements imposed by the current project, i.e. single word transmission from Linux to AUTOSAR and stream transmission from AUTOSAR to Linux.

3.5 Modifications to the Linux Kernel

One of the goals of this setup was to keep the modification of the Linux source code to a minimum. Hence, additional functions were implemented in a kernel module, where possible. This module can be compiled into the kernel or loaded at runtime. Loading of the module has no impact on the timing behaviour of the real-time parts, because no shared resources are needed. Some additional modifications were still needed to allow a separation of the operating systems.

The Linaro Linux Kernel for an OMAP4460 SOC contains a complete hardware description of all submodules on the chip. This description is read during startup and all hardware modules are automatically initialized by the kernel. No dynamic configuration of the SOC hardware is performed. This initialization also includes all hardware modules which were shared or statically assigned to AUTOSAR. As this additional initialization would undo all previous initializations done by AUTOSAR, we had to reduce the hardware initialization. For this it was sufficient to remove any module used by AUTOSAR from the hardware description tables provided in the Linux source tree.

During it's initialization routine Linux on the OMAP4460 also configures the GIC and sets all interrupts to target the first core used by Linux. As the GIC is already set up by AUTOSAR this additional initialization code was removed. Instead the interrupt targets are set when the interrupt is activated within Linux. The code for setting the target upon activation was also added.

In the SRAM management code it was completely sufficent to set the SRAM size to zero. The current Linux kernel already includes code paths to skip all SRAM initializations in case there is no SRAM on the current system and to reject all allocations on the SRAM.

As these changes are quite small and only include parts of the kernel which change rarely, it is easily possible to re-apply the same patches to newer kernel versions, making updates of the Linux portions simple.

4. CONCLUSIONS AND FUTURE WORK

We were able to show that it is possible to use separate cores for different operating systems on multi-core processors. While previous work [8] only showed such a setup for x86 systems running two invocations of the same Linux system, we extended this work to the ARM-Platform where one of the operating systems was a real-time operating system. The real-time ability was not impaired by the simultaneously running non-real-time system.

Future work may include a better resource sharing among both systems. As it is rare for both systems to require the same hardware at the same time, it may be possible to provide a suitable protocol that enables the necessary resource-sharing between systems. However such a protocol would have to provide guaranteed latency for the real-time system, but not for Linux system, so an asymmetric protocol could be used. This could for example be implemented using Mixed Criticality Locks [12], which may be adapted for multicore architectures.

Also a detailed comparison and experimental evaluation of the different methods for implementing mixed criticality systems is still needed. At this point however such a comparison would be beyond the scope of this article.

References

[1] *ARM Generic Interrupt Controller - Architecture Specification*. ARM. 2011.

[2] Jan Altenberg. *Fastboot Technologie für Linux*. Tech. rep. Mülhofen, Germany: linutronix, 2010.

[3] *Arctic Core - the open source AUTOSAR embedded platform*. ARCCORE AB, 2011. URL: `http://www.arccore.com/products/arctic-core/`.

[4] *Avionics Application Software Standard Interface*. ARINC Report 653. Aeronautical Radio Inc., Jan. 1997.

[5] Michael Christofferson. *4 Ways to Improve Linux Performance for Multicore Devices*. IEEE Spectrum Online Tech Insider Webinar. 2013.

[6] *Cortex^{TM}-A9 MPCore - Technical Reference Manual*. ARM. 2009.

[7] Wolfgang Denk. *Das U-Boot*. 2012. URL: `http://git.denx.de/cgi-bin/gitweb.cgi?p=u-boot.git;a=summary`.

[8] Adhiraj Joshi et al. "Twin-Linux: Running independent Linux Kernels simultaneously on separate cores of a multicore system". In: *Proc. of the Ottawa Linux Symposium*. Ottawa, Canada, 2010, pp. 101–107.

[9] *OMAP4460 Multimedia Device Silicon Revision 1.x. Technical Reference Manual*. Version F. Texas Instruments. 2011.

[10] *Road vehicles – Functional safety*. ISO/FDIS 26262. International Organization for Standardization, Dec. 2010.

[11] Steven Rostedt and Darren V Hart. "Internals of the RT Patch". In: *Proc. of the Ottawa Linux Symposium*. Vol. Two. Ottawa, Canada, 2007, pp. 161–171.

[12] Jörn Schneider. "Overcoming the Interoperability Barrier in Mixed-Criticality Systems". In: *19th ISPE International Conference on Concurrent Engineering - CE2012*. Trier, Germany, 2012.

A Constraint-Based WCET Computation Framework

Hajer Herbegue Mamoun Filali Hugues Cassé

CNRS IRIT
Université de Toulouse, France
first_name.last_name@irit.fr

ABSTRACT

OTAWA is a tool dedicated to the WCET computation of critical real-time systems. The tool was enhanced in order to take into account modern micro-architecture features, through an ADL-based approach. Architecture constraints are expresses such that they can be solved by well known efficient constraint solvers. In this paper, we present how we could describe some complex architecture features using the Sim-nML language. We are also concerned by the validation and the animation point of views.

1. INTRODUCTION

OTAWA is a tool dedicated to the analysis of critical real-time systems. The worst case execution time (WCET) computation is one of the crucial analyses it provides. In a previous paper [12], we have shown how to enhance this tool in order to take into account modern micro-architecture features and complex instruction set architectures. An approach based on architecture description language (ADL) for the execution time analysis considers the architecture description in the Sim-nML language [11]. We have surveyed how to express architecture constraints such that they can be solved by well-known efficient solvers. In this paper, we present an extension of the Sim-nML language that allows to handle a set of complex features of modern processors. We are also concerned by a validation point of view. Since the validation of almost all currently used solvers is out of reach, we extended the OTAWA framework to allow the validation of the results returned by these solvers. This approach is not new: it is already used in assistant theorem provers [7, 16] where results provided by other less reliable but more efficient tools [2] are established again, i.e., validated. The inclusion of the validation layer is a novelty of OTAWA since it represents an easy and fast way to verify if analysis results meets the initial architecture specification.

The rest of our paper is organized as follows: Section 2 is an overview of the related works. Section 3 recalls the context of our work. We present the OTAWA tool and the constraint-based approach for WCET computation. Section 4 presents the Sim-nML language and its extension to describe some complex features of modern processors. Section 5 presents another contribution of the paper related to validation and animation aspects. Section 5 draws some conclusions.

2. RELATED WORKS

Over the last years, many studies have been undertaken with respect to WCET computation for pipeline architectures. Among them, we mention the work of [15] concerned by verifying structural properties related to the wellformedness of the architecture. A graph-based model of the processor is generated from an ADL-based description. Architecture structural properties are validated using algorithms applied on the graph. With respect to dynamic aspects, the model checking approach has been experimented with mitigated results [9]. The basic timed automata model enhanced with game theory aspects [4] seems promising. Nevertheless, abstract interpretation based approaches [19] are today well established. In this paper, we are interested in the AI based approach on constraint satisfaction [6] and its validation.

3. ADL-BASED APPROACH FOR WCET COMPUTATION

OTAWA [3] is a framework dedicated to WCET computation founded on an abstraction layer that describes the target hardware and the instruction set architecture (ISA), as well as the binary code under analysis. The instruction set architecture (ISA) specification is expressed in the Sim-nML language. The hardware architecture is described through an XML format. Yet only some architectures can be handled by such a model. A pipeline analysis consists in modeling the execution of basic blocks[1] on the pipeline and then computing the corresponding execution costs [18]. A constraint-based approach, presented in [12], is based on ADL processor descriptions and uses constraint specification languages and resolution methods to compute the time cost of a basic block. The target processor is described using the Sim-nML language [10, 17]. In addition to the ISA level, the architecture description includes the hardware components and the execution model of instructions. Sim-nML was extended to support such a description of the processor. The carried analysis aims to estimate the time cost of basic blocks of a program. Using the processor description in the Sim-nML language and the basic block, we generate a constraint-based description. The execution time of a basic block is described as a Constraint Satisfaction Problem (CSP) [6]. This approach handles complex processors with out-of-order execution, superscalar stages, pipelined functional units, etc. This is done within an automated work flow, presented in Figure 1.

4. THE SIM-NML LANGUAGE

[1]A basic block is a sequence of instructions, without any branch, which makes up the execution path of a program.

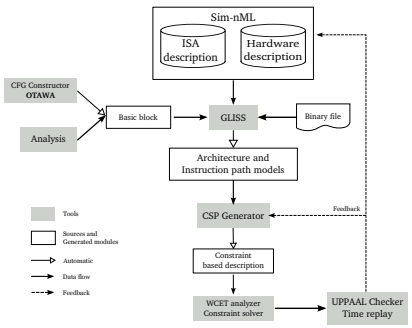


Figure 1: ADL-based work flow for WCET analysis

A representation of an architecture consists of the description of its hardware components and the supported instruction set. Sim-nML [11] is a hierarchical and a highly structured language able to perform such a description. From this, it provides the ability to generate processor specific tools. In Sim-nML, the processor model is described at instruction level, as a hierarchical structure using an attributed grammar. The instructions and the addressing modes are described by pre-defined attributes. The *syntax* attribute defines the assembly representation of the instruction. The attribute *image* gives the binary representation and the attribute *action* defines the semantics of the instruction. See lines 15-18 of listing 1.

Listing 1: Sim-nML processor description

```
1  stage FE , DE , IS , ALU[2] , MEM , CM
2  extend FE , DE , IS , CM
3      capacity = 2          // degree of super-
           scalarity
4      inorder = true        // in-order stages
5  extend ALU , MEM
6      inorder = false // out-of-order stages
7
8  // Fetch Buffer and Re-order Buffer
9  buffer FBuf [4] , RoB [8]
10
11 reg PC [1,card(32)] // 32-bit PC register
12 reg R [32,card(32)] // 32 registers of 32 bits
13 mem M [32,card(8)]// a memory of 2^32 8-bit words
14
15 op add (d:card (2),s1:card (2),s2:card (2))
16  syntax = format ("add r%d r%d r%d",d,s1,s2)
17  image = format ("00%2b%2b%2b",d,s1,s2)
18  action = {R[d] = R[s1] + R[s2] ;}
19  uses = FE & FBuf,DE,IS & RoB,ALU[0] & R[s1].read
        & R[r2].read & R[d].write & RoB #{1}, CM
20
21 op load (d:card (2),s:card (2))
22  ...
23   uses = FE & FBuf,DE,IS & RoB,MEM & R[d].write &
          R[s].read & M.read & RoB #{10}, CM
```

We extended the Sim-nML language such that we can declare the hardware structure of the processor[2]. Precisely, the extended language provides the syntax to define the pipeline stages, the resources accessed by the instructions when they execute on the pipeline. The properties of these hardware components are specified as attributes. So, we can declare stages, buffers, registers and memories as hardware resources. Lines 1-13 of Listing 1 describes the processor in the Figure 2. The instruction definition is extended with an attribute *uses* that describes the execution model of the

[2]Extensions are underlined in the listings.

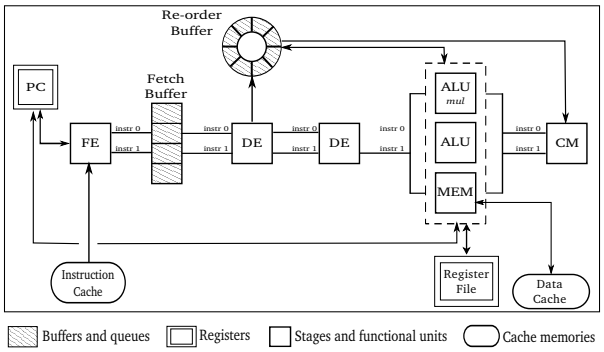


Figure 2: An out-of-order superscalar processor

instruction. The execution model represents the instruction behavior in terms of resources allocation. For example, to begin executing on a stage, an instruction has to wait for its resources to be available. Therefore, the execution time of an instruction is impacted by the general resources state. The *uses* attribute defines, in a timed sequence called clause, the resources used by an instruction in each step of its execution. A sequence is defined using commas. Every clause in a sequence represents a step of the instruction execution. In every step, one or more resources are required, and access can be in a read or a write mode. Parallel access is expressed by an operator &. Access to some resources can take a fixed duration t that can be specified as $\#\{t\}$. An example of *uses* attribute is given in lines 19 and 23 of Listing 1.

Listing 2: Specialized execution with different latencies

```
1 stage FE , DE , IS , ALU[2] , MEM , CM
2 extend DIV
3  uses = FE , DE , IS , ALU[1] & R[rn].read &
4         R[rd].write & R[rm].read #{25} , CM
5 extend MUL
6  uses = FE , DE , IS , ALU[1] & R[rd].write &
7         R[rm].read & R[rn].read #{5} , CM
```

Listing 3: Multiple load instruction

```
1 extend load_multiple
2  uses = FE , DE , IS ,
3   (if reglist<0..0> == 1 then MEM & M.read & R
        [0].write .. endif),
4   (if reglist<1..1> == 1 then MEM & M.read & R
        [1].write .. endif), ... , CM
```

The language was extended to handle some complex features of recent architectures. In fact, modern architectures present complex pipelines and instructions with complex execution models. We were able to handle pipelines with specialized execution units, micro-coded instructions and pipelined functional units (like floating point pipelines). For example, we assume having a pipeline with 2 out-of-order ALU units, among which only one executes multiplication instructions. This feature is relevant in execution time computation. Multi-cycle instructions and the micro-coded instructions as the load/store multiple are handled by our description language. Listing 3 presents a load multiple instruction where the *reglist* parameter is a bit sequence. A

bit is set to one if the corresponding register is loaded. Different latencies can be specified for every stage and are taken into consideration when generating temporal constraints. The example of Listing 2 considers the same architecture in Figure 2. We assume that multiplication and division instructions are executed by the second *ALU* unit, which is the specialized functional unit. However, different latencies are specified for the two instructions on that functional unit (see Listing 2). These clauses, specified for every instruction supported by the processor, will be used, with the stages attributes to generate the instructions constraints [12]. In fact, we use temporal intervals to represent the lifetime of instructions on the pipeline stages and the resource allocations. The instruction dependencies within a basic block are expressed with constraints on the time intervals. The constraint description captures the architecture and instruction semantics such as the resource allocation strategy, the data dependencies, the structural dependencies, contentions on shared resources, etc. The constraints are combined to formulate a CSP, which resolution provides the time cost of basic blocks of a program.

5. VALIDATION AND ANIMATION

The aim of the OTAWA tool is to provide an environment for the hardware architect. In this section, we consider two tools related to validation and animation. These tools are based on an internal representation modeling the architecture and the instructions behavior.

ISA level tasks:	**Step level tasks:**
$\langle \text{ISA} \rangle^{\langle \text{interval} \rangle}_{\langle \text{instruction} \rangle}$	$\langle \text{Step} \rangle^{\langle \text{interval} \rangle}_{\langle \text{instruction} \rangle, \langle \text{step} \rangle}$
Basic tasks (leaves):	
$\langle \text{occurrence}_{\mid ?} \rangle \langle \mathit{Stage} \rangle^{\langle \text{interval} \rangle}_{\langle \text{instruction} \rangle, \langle \text{step} \rangle} \mid$	
$^{[r \mid w]}_{\langle \text{occurrence}_{\mid ?} \rangle} \langle \mathit{Resource} \rangle^{\langle \text{interval} \rangle}_{\langle \text{instruction} \rangle, \langle \text{step} \rangle, \langle \text{index} \rangle}$	
$\langle \mathit{Stage} \rangle$, $\langle \mathit{Resource} \rangle ::= \langle \mathit{Register} \mid \mathit{Buffer} \mid \mathit{Memory} \rangle$	

Table 1: Collected clauses syntax

5.1 The internal representation

Since we are concerned by modeling concurrency of instructions, we have chosen an OCCAM based representation [8, 13]. We consider the following basic constructors:

- `USES` : This is the terminal case in which a basic resource request and an access are specified.
- `SEQ` : This is a sequential behavior. Each element of this sequence is specified recursively by a clause. Intuitively, the `SEQ` constructor will allow us to specify the execution path of an instruction. Each clause of this path will specify the local behavior with respect to a stage of the processor, what we called a step.
- `PAR`. This is a concurrent behavior. Each element is specified recursively by a clause. Intuitively, we express as such the simultaneous use of resources during a step.
- `ATTR`. These are general attributes superposed to a clause, e.g., timing ones. Actually, our internal representation is decorated with the resolved constraints.

Our internal representation is based on generic attributed clauses. For our validation purposes, we instantiate the attribute type as the corresponding time interval. With respect to our concerns, the collected clauses can be described through the light DSL (Domain Specific Language) given by the syntax in Table 1. For instance, if we consider the load instruction : `ldr r3 , [ r11 , -#20 ]` executed on the architecture of Listing 1, the following clause represents the effective resources access of the instruction.

$$i0\_clause = {}_0FE^{[0,1]}_{0,0} \ \& \ {}^{?}_{?}FBuf^{[0,1]}_{0,0} \ \& \ {}^{r}_{15}R^{[0,1]}_{0,0} \ , \ {}_0DE^{[1,2]}_{0,1} \ ,$$
$${}_0IS^{[2,3]}_{0,2} \ \& \ {}^{?}_{?}RoB^{[2,3]}_{0,2} \ , \ {}_0MEM^{[3,4]}_{0,3} \ \& \ {}^{r}_{0}M^{[3,4]}_{0,3} \ \& \ {}^{r}_{11}R^{[3,4]}_{0,3}$$
$$\& \ {}^{w}_{3}R^{[3,4]}_{0,3} \ \& \ {}^{?}_{?}RoB^{[3,4]}_{0,3} \ , \ {}_0CM^{[4,5]}_{0,4}$$

5.2 Validation.

We have studied two kinds of validation:

- *Instruction constraints validation.* It consists in checking that the results are coherent with respect to the initial representation. For example, we validate that the intervals of instruction steps respect the data hazard constraints.
- *Architecture validation.* It consists in checking that the results are coherent with respect to the studied architecture. For instance, we validate that, in an in-order pipeline, an instruction steps occurs before its successors.

$$\forall \ s^{[l,u)}_{i,s} \in STEP. \ \forall \ s^{[l',u')}_{i',s'} \in STEP. \ i < i' \Rightarrow u \leq l'$$

Although, theoretically these validations should not be necessary, our experiments have shown that they are of great help. Actually, it is much easier to assess these validations than those on the usual execution graphs [14] that can be huge.

5.3 Animation.

The aim of the "animated" views is to assist the architect to better understand instruction behaviors. For that purpose, we consider a well known formal model: that of timed automata [1] for which verification tools like UPPAAL [5] implement the decision procedure. As a matter of fact, the UPPAAL framework is by now a mature tool which offers a powerful simulator in order to interact dynamically with the architecture under study. Currently, we generate automatically the *instruction view* and the *stage view*. Also, architects can use the UPPAAL query language to express temporal predicates. Then, such predicates will be decided automatically. Moreover, if some property is not verified a counter example is exhibited. To summarize, the architect user can step along the execution of his model and validate general dynamic properties.

Behavior specification: from timed clauses to timed automata.

Basically, to each clause, we associate a timed automata location. Thanks to an invariant, control remains in such a location starting from the lower bound until the upper bound of the interval associated to the clause is reached. A guard ensures that such a location is not left before the upper bound is actually reached. Each of our animated views consists in a network of such automata sharing a global clock `clk`. Last, since our automata progresses according to time (not to internal events or synchronizations), the labels of their states are also meaningful.

- The instruction view. In this view, a timed automata was assigned to each instruction. Stepping trough this view allows us to see how each instruction evolves. This view is especially interesting for observing the relative "speed" of each instruction: when an instruction enters the pipe and maybe stalls over its successive stages.
- The stage view (Figure 3). In this view, to each stage is associated a timed automata. Stepping through this view allows us to see how stages evolve. This view is especially interesting for observing stages occupancy.

Figure 3: Stage view network timed automata

6. CONCLUSION

In this paper, starting from our extension of the OTAWA tool allowing WCET computations for today architectures [12], we have extended the Sim-nML language in order to handle modern processors features. We found the constraint-based time computation method suitable for expressing complex instruction features. We have also presented a light DSL for expressing architecture properties. Last, we have considered how to validate and animate the results obtained through constraint solvers. As a future work, we intend to use the DSL presented in this paper to formalize the architecture specification and constraints, in order to elaborate a reliable description of the architecture constraints.

7. REFERENCES

[1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Comput. Sci.*, 126(1):183–235, February 1994.

[2] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of sat/smt solvers to coq through proof witnesses. In *CPP*, pages 135–150, 2011.

[3] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In *Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, 2010.

[4] J.-L. Béchennec and F. Cassez. Computation of wcet using program slicing and real-time model-checking. *CoRR*, 2011.

[5] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *QEST*, pages 125–126, 2006.

[6] N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12(1):21–62, Mar. 2007.

[7] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.

[8] A. Burns. *Programming in Occam 2*. Addison-Wesley, 1988.

[9] A. Dalsgaard, M. Olesen, M. Toft, R. Hansen, and K. Larsen. METAMOC: Modular execution time analysis using model checking. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2010.

[10] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nml. European Design and Test Conference (EDTC), 1995.

[11] M. Freericks. The nml machine description formalism. Technical Report 1991/15, TU Berlin, 1991.

[12] H. Herbegue, H. Cassé, M. Filali, and C. Rochange. Hardware architecture specification and constraint based wcet computation. In *International Symposium on Industrial Embedded Systems (SIES)*, June 2013.

[13] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[14] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, 2006.

[15] P. Mishra and N. Dutt. Modeling and validation of pipeline specifications. *ACM Trans. Embed. Comput. Syst.*, pages 114–139, Feb. 2004.

[16] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Number 2283 in Lecture Notes in Computer Science. Springer, 2002.

[17] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *International Conference on VLSI Design*, 2000.

[18] C. Rochange and P. Sainrat. A context-parameterized model for static analysis of execution times. *Transactions on High-Performance Embedded Architectures and Compilers II*, 2009.

[19] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 2008.

Taming Control Exchange for Software Defined Radio in System Level Models

Andrea ENRICI, Ludovic APVRILLE, Renaud PACALET
Institut Mines-Telecom, Telecom ParisTech
CNRS/LTCI, Sophia Antipolis, France
Email: firstname.lastname@telecom-paristech.fr

***Abstract*—A great amount of the complexity encountered when dealing with SDR architectures comes from simultaneously facing interwoven dataflow and controlflow requirements at system level [1]. Failing to consider both requirements when programming SDR systems leads to poor performance and inefficient utilization of the hardware resources. In this paper we take a UML-based dataflow Model Driven Design methodology for SDR applications and enrich it with novel expressiveness to capture controlflow, early at modeling phase, by means of intuitive Domain Specific Modeling Languages. Our contributions provide the necessary features to tame control needs and constraints in high-level models from which the executable control code can be automatically generated by means of synthesis mechanisms.**

I. Introduction and Context

Current radio systems (e.g. base stations) have to support multiple communication standards in order to take advantage of the growing number of available air-services (e.g. 3G, GPS). The latter share numerous Digital Signal Processing algorithms demanding an architecture capable of implementing the corresponding computations in the most generic and flexible way.

Software Defined Radio [2] is perceived as the technology that could solve this flexibility issue by implementing most of the complex signal processing operations of radio receivers and transmitters via software instead of dedicated hardware. A SDR system is formed by DSP processors governed by (heterogeneous) general purpose processor(s) and is configurable via software: thus, multiple applications (*waveforms*) are supported by pooling the architecture (*platform*) computational, storage and communication resources.

SDR equipment design is a hw/sw co-design topic that deals with most issues of design space exploration for embedded real-time systems. In fact, the required properties of SDR design share many commonalities with other fields, e.g. image and video processing, thus allowing solutions specific to SDR systems to be applied to other domains. In these domains, MDD methodologies have been widely used as they expose and exploit high-level structures and interactions of systems. In particular, dataflow MoCs are popular in the design and implementation of signal processing systems due to their intuitive match with communication system block diagrams and the formal structure exposed by such representations [3].

So far, many dataflow-based methodologies have been applied to SDR systems. [4] permits executable code generation from an initial dataflow model through a series of graph transformations. [5] proposes a lightweight approach facilitating cross-language, cross-platform migration and prototyping for different signal processing domains, including SDR. [1] describes a dataflow methodology based on UML, capable of generating executable code from high-level abstract models.

However, one must keep in mind that SDR systems are not only dataflow oriented: methodologies also have to deal with control issues and above all with the conflicts that may arise between the two flows. Some SDR architectures exhibit conflicts when transferring control and data items due to shared resources, [6]. Others, [7], have multiple Control Units with different set of control features. In the first case, contention becomes an issue and deadlocks may occur. In the second case, mapping of tasks over the platform is constrained by the control capabilities of a single CU. In either case, if controlflow is not taken into account early at modeling phase, a MDD methodology may generate code with poor performance and sub-optimal scheduling.

In this paper we introduce a novel approach based on DiplodocusDF [1] that solves the above issues. In contrast to purely dataflow-based methodologies, our solution explicitly describes control in architecture and application models, thus enabling description of complex scheduling scenarios and mapping constraints that do not only depend on the way data flows through processing operations. We start with a brief introduction to DiplodocusDF and the platform over which it has been employed. Next, the problem with controlflow in SDR systems is presented by a case study which is the starting point for a discussion illustrating our solution. Finally we outline how our contributions help solving the issues presented by the case study and conclude with the state and directions of future works.

A. The Hardware Platform Embb

[6] proposes a new generic baseband architecture for SDR applications. An instance of such a platform is depicted in Fig. 1. It is composed of (1) DSP units, (2)

a system interconnect and (3) a main CPU. The DSP units are in charge of executing processing operations (e.g. FFT). They are equipped with a hardware accelerator as computational unit (Processing Sub-System, *PSS*), a DMA to transfer data, an internal memory mapped on the main processor memory and a microcontroller (μC) that allows to reduce interventions of the main CPU. The latter is linked to DSPs via the system interconnect that allows data-blocks and control information to be exchanged among units. The main processor executes the waveform control operations: it manages data-transfer operations, the computational units and the interface with the external environment (Fig. 1, yellow area).

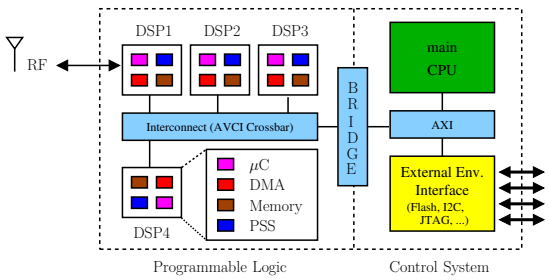


Fig. 1: Architecture of an Embb instance

Since the processing operations of a waveform pool data storage, computation and transfer resources, Embb provides means for its units to exchange control information. More specifically: each DSP unit has several interrupt lines: one input, and multiple output and internal lines dedicated to event signaling; PSSs, DMAs and μCs are equipped with control and status registers. Interrupts are raised by PSSs upon processing completion, by DMAs upon data-transfer completion and by μCs or by the main CPU software to mutually signal events.

B. The DiplodocusDF Methodology

DiplodocusDF [1] is a UML dataflow-oriented MDD methodology for the domain of SDR applications, Fig. 2. It stems from DIPLODOCUS, [8], a UML Model Driven Engineering methodology for hw/sw partitioning of SoC at high abstraction level, currently implemented by the free software TTool [9]. The core strength of DIPLODOCUS is the automatic transformation of models for simulation and formal verification [10]. However, the DIPLODOCUS approach is too abstract to permit automatic code generation for SDR systems as models lack the necessary expressiveness to face the complexity of platforms and waveforms.

DiplodocusDF is a first attempt to fill this gap; it enriches DIPLODOCUS with the following extensions:

1) **A dataflow semantics**: an application is a dataflow graph where nodes represent *tasks* (processing, routing, addressing operations) interconnected by special links called *signals* used to carry data-blocks and data parameters (e.g. r/w memory addresses).
2) **A specialization of the architecture language**: SDR platforms are represented as a network of computation nodes (e.g. DSPs), storage nodes (memories) and data-transfer nodes (bus, bridges, DMAs) connected by data links.
3) **A code generation environment**: the description of a waveform mapped over a platform is translated in C language via an intermediate representation enriched by the platform API and by a static scheduler (*RTE*).

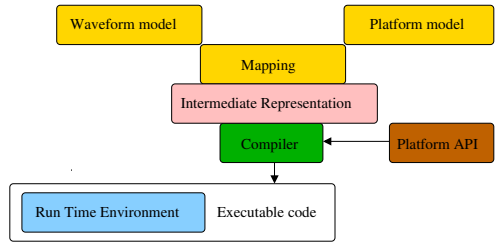


Fig. 2: The DiplodocusDF methodology

II. Dataflow vs Controlflow

In Embb conflicts between data and control can be found in the bridge that separates the programmable logic domain from the control system domain, Fig. 1. This unit equally serves read/write requests from both domains, regardless their type (control or data): when the bridge is busy with a DMA transfer, all control accesses are blocked. Fig. 3 depicts such a scenario: task T1 processes data that the RF interface stores in memory MEM and transfers them via DMA to T2 for further processing (dotted line). Parallel to this, task T3 runs on the CPU and upon execution completion triggers T4, running on DSP2 (continuous line).

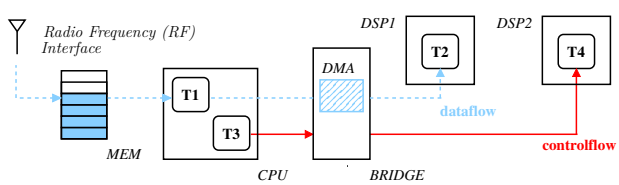


Fig. 3: Dataflow vs controlflow in Embb

Due to the underlying MoCs, dataflow MDD methodologies ([1], [4]) usually schedule operations statically and only according to availability of input data to tasks. Fig. 4 shows how such an approach for the scenario of Fig. 3 leads to a deadline miss. Because of availability of input data from the RF interface, T1 is scheduled for execution first and accesses the bridge before T3 can trigger T4. As a consequence the DMA transfer locks the bridge and prevents control information to reach DSP2 on time causing T4 to miss its deadline!

Of course, approaches that modify the scheduler only may

be implemented, but we are not concerned with them as we look for a higher level solution based on system models.

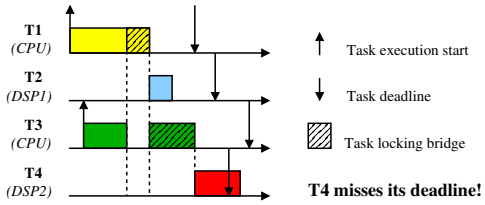


Fig. 4: Typical scheduling of dataflow-based approaches

DiplodocusDF Limitations:

DiplodocusDF's approach to controlflow is rigidly dominated by its inner dataflow nature. In the most general case (processing operations), tasks consist of up to two UML Activity Diagrams (*ADs*): one for modeling a task configuration and one for modeling a task execution. Within a task's ADs, controlflow is implemented by means of the usual structures (e.g. conditional statements, loops) but the data-block parameters (e.g. data offsets, addresses) are the only control variables that can be tested to perform different actions. Additionally, none of the two ADs is allowed to communicate and exchange the data-block parameters with other tasks, independently of a data-block's dataflow itself. Consequently, data-block parameters are strictly scoped to the tasks to which a data-block is connected to and no primitives are available to exchange them to other tasks. The latter are, in fact exclusively linked by data-blocks. As a consequence, the application description is limited to a network of data-dependent operations that can only be scheduled according to decisions taken on the I/O availability of data. Coherently with the above expressive power for waveforms, platform models provide no medium to describe how control information is exchanged among the architecture units. At mapping phase, this means that tasks are blind with respect to the real controlflow capabilities of the platform. All the above-mentioned lack of expressiveness may thus lead to sub-optimal resource utilization, sub-optimal scheduling and performance as illustrated in the case study of Fig.3 and Fig.4.

III. Controlflow in System Level Models

Following the previous discussion, we describe our solution to enrich DiplodocusDF in order to model controlflow and the conflicts with dataflow in both architecture and application graphs, Fig. 5.

In the waveform graph, Fig.5(a), we take advantage of the control exchange primitives *events* and *requests* from DIPLODOCUS. These primitives have already been described in detail in [10]; therefore, we simply recall their semantics. Events are synchronous point-to-point unidirectional primitives between two tasks, used to synchronize the execution of ADs. They are stored, between sender and receiver, in an intermediate FIFO (finite or infinite), operators are provided to tasks to send, receive and test for the presence of events. Requests are asynchronous multipoint-to-point unidirectional primitives between tasks, used to spawn execution of ADs. They are stored in an infinite FIFO between sender and receiver; tasks are provided with operators to send a request and to retrieve its arguments upon reception of a request. Both events and requests are in fact able to carry arguments (e.g. counters, flags). **In addition to the above primitives, we add a new class of requests, *notified requests*, and a new class of nodes, *fork and join nodes*, for exchanging events among tasks**. A *notified request* is a request provided with a notification mechanism sent by the receiver task upon completion of its execution, to the request sender. This new class of requests carries integer parameters (e.g. error codes) like standard requests. Two new operators, one for sending the notification and one for receiving it as well as retrieving its parameters, are provided. *Fork and join nodes* for events are a new class of routing nodes for events only. A fork node allows events from one source task to be dispatched to multiple destination tasks, in parallel and simultaneously (broadcast). A join node allows multiple events from different tasks to be serialized and dispatched to a single receiver task (gathercast); serialization takes place according to priority assigned to single events. For the latter purpose events are enriched so that they can be tagged with a priority.

The core strength of our novel approach lays in abstracting and modeling control information exchange in the architecture also. On top of enriching the models' expressive power by introducing new modeling features, our solution includes all the control-related architecture constraints that limit mapping of an application onto a platform.

In the architecture graph, Fig.5(b), we introduce a new class of links called *control links* and a new class of nodes called *control join nodes*. A *control link* is an oriented point-to-point link in the architecture graph between two units, regardless their type (control or execution unit). As opposed to DiplodocusDF data links, control links are specifically instantiated to transfer control information only. Moreover, while data links model physical paths in the architecture through which items are routed and transferred in the hardware, control links are virtual. They are an abstraction used to model the control exchange flow paths in the architecture, thus there is not always a one-to-one correspondence with a physical path in the hardware through which control signals and variables are transferred. Given a source unit S and a destination unit D, a control link is instantiated iff control information can be exchanged between S and D regardless the physical medium over which the transfer effectively takes place (e.g. dedicated interrupt line, bus).

A *control join node* is the equivalent of an application

join node for events, at the architecture level. It allows serial reception of multiple control items, passing through several control links that come from several different input units (gathercast). This particular kind of node is used to model hardware mechanisms like interrupt controllers, but can also be instantiated to take into account software control mechanisms implemented by the platform OS.
Additionally, we extend the semantics of bridges in order to consider control links and the control items they carry. So, bridges become nodes that route both control and data information coming from control and data links, when the two flows need access to shared resources as highlighted in the case study. The semantics of buses need not be extended as it is not required to implement a different policy when transferring data or control items.

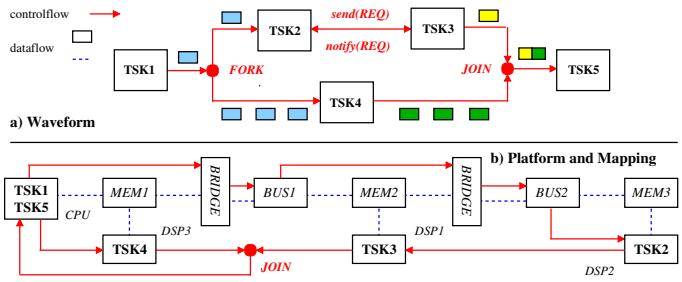


Fig. 5: Taming controlflow in a sample SDR system

In order to automatically generate the control code, information contained in the system models must be merged by establishing a relation between an element of the waveform and an element of the platform. In addition to DiplodocusDF relations where signals are associated to data links and tasks to processing/control units, **we eventually complete the mapping phase by projecting events and requests from the application to control links and control join nodes in the architecture, Fig. 5(b)**.

IV. Expected Results and Conclusions

By means of the contributions described in section III, our enriched version of DiplodocusDF is now equipped with the correct expressive power to describe controlflow and to enable scheduling of tasks in a more flexible way. Fig. 6 shows how the scenario of Fig. 3 can be executed differently if the separation between data and control that we introduced at system-level models is maintained throughout the whole methodology. This separation of concerns leads to implementations where the executable code is automatically generated with independent data and control tasks.

In our case study, this translates at scheduling level, into task T1 being divided into two tasks. The new task T1 now only implements the processing of data that are then dispatched to T2 by a standalone task T0 in charge of the cross-domain DMA transfer. So, the conflict between control and data for accessing the bridge is solved and a

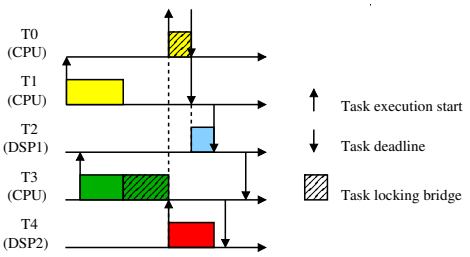


Fig. 6: A solution to conflicting control and data flows in Embb

more performing scheduling solution can be implemented as illustrated.

We are currently working at integrating the described enhancements in the DiplodocusDF methodology as well as implementing these extensions in TTool. Future works will address formal verification, code generation and scheduling. Concerning the former, we envisage to take advantage of the formal semantics already available in DIPLODOCUS and enrich it to consider the new modeling features we discussed in this paper, before and after mapping. On the side of code generation, we will provide new elements to decorate the models so that user will be able to specify how the executable code is implemented. For instance, user will be able to choose whether to implement the control code corresponding to events and requests as interrupt-based or polling-based, in order to tackle performance aspects. Lastly, we will work on scheduling algorithms and strategies that fit the models' new expressive power, providing policies to properly handle conflicts between dataflow and controlflow as illustrated in the above case study and its proposed solution.

References

[1] J. M. Gonzalez Pina, "Application Modeling and Software Architectures for the Software Defined Radio," Ph.D. dissertation, Télécom-ParisTech, May 2013.

[2] J. Mitola III, "Cognitive Radio: An Integrated Agent Architecture for Software Defined Radio," Ph.D. dissertation, Royal Institute of Technology (KTH), May 2000.

[3] S. S. Bhattacharyya, "Hardware/Software Co-synthesis of DSP Systems," in *Programmable Digital Signal Processors: Architecture, Programming, and Applications*, 2001, pp. 333–378.

[4] C. Moy and M. Raulet, "High-Level Design for Ultra-Fast Software Defined Radio Prototyping on Multi-Processors Heterogeneous Platforms," *Advances in Electronics and Telecommunications*, vol. 1, no. 1, pp. 67–85, 2010.

[5] C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya, "A Lightweight Dataflow Approach for Design and Implementation of SDR Systems," in *SID-WInComm*, 2010.

[6] N.-u.-I. Muhammad, R. Rasheed, R. Pacalet, R. Knopp, and K. Khalfallah, "Flexible Baseband Architectures for Future Wireless Systems," in *EUROMICRO DSD*, 2008, pp. 39–46.

[7] "Parallela," http://www.parallela.org/.

[8] L. Apvrille, W. Muhammad, R. Ameur-Boulifa, S. Coudert, and R. Pacalet, "A UML-based Environment for System Design Space Exploration," in *IEEE ICECS*, 2006, pp. 1272–1275.

[9] "TTool," http://ttool.telecom-paristech.fr/.

[10] D. Knorreck, L. Apvrille, and R. Pacalet, "Formal system-level design space exploration." *Concurrency and Computation: Practice and Experience*, vol. 25, no. 2, pp. 250–264, 2013.

Improved Priority Assignment for the Abort-and-Restart (AR) Model

H.C. Wong and Alan Burns
Real-Time Systems Research Group, Department of Computer Science, University of York, UK.
{hw638, alan.burns}@york.ac.uk

Abstract

This paper addresses the scheduling of systems that implement the abort and restart (AR) model. The AR model requires that preempted tasks are aborted. As a result high priority tasks run quickly and shared resources need not be protected (as tasks only work on copies of these resources). However there is significant wastage as low priority tasks may be subject to a series of aborts. We show that exact analysis of the AR model is intractable. A sufficient but tractable test is developed and is used to address the priority assignment issue. Again an optimal tractable algorithm is not available. The paper develops a priority assignment heuristic that is demonstrated to perform better than existing schemes.

1 Introduction

Abort-and-Restart is a scheme to support Priority-based Functional Reactive Programming (P-FRP). P-FRP has been introduced as a new functional programming scheme [2] for real-time systems. It combines the property of atomic execution from Functional Reactive Programming (FRP) [11], and supports priority assignments. To achieve this property of P-FRP, preempted tasks are aborted and the tasks restart as new once the higher priority tasks are completed. We call it the Abort-and-Restart (AR) model in this paper.

Various forms of priority inheritance and priority ceiling protocols have been developed [10] to deal with the problems of shared resources. The AR model does not face the problems because tasks do not access resources directly. But the disadvantage is that aborted tasks delete the old copy of the resource and restart as new, hence the time spent before preemption is wasted. In this paper, we call this wasted time, the *abort cost*.

1.1 Motivation for the AR model

The AR model provides strong correctness guarantees on dealing with shared resources, and it also supports FRP which has been used for the domains of computer animation, computer vision, robotics and control systems [6]. Original FRP cannot be used for real-time systems but P-FRP has rectified this. Preemptible Atomic Regions (PAR) is a new concurrency control abstraction for real-time systems [8]. The basic notions of the AR model and the PAR model are similar. In other words, the AR model has been implemented in a common programming language.

1.2 Contributions

This paper presents an analysis for the AR model. It first confirms that an exact analysis is not tractable as the critical instance cannot, in general, be identified in polynomial time. The second contribution is to develop a new schedulability test for the AR model. A final contribution is to address priority assignment. General priority assignment policies such as rate and deadline monotonic, are not optimal for the AR model or the developed test. In this paper, we evaluate a number of existing priority assignment policies and provide an improved (though still not optimal) policy called *Execution-time-toward-Utilisation Monotonic* (EUM).

2 System Model and Related Work

We consider the static priority scheduling of a set of sporadic tasks on a single processor. Each task gives rise to a potentially unbounded sequence of jobs. The notations and formal definitions used are as follows: N is the number of tasks. τ_i, any given task in the system. C_i worst-case execution time (WCET), T_i period, D_i deadline, P_i priority, R_i worst-case response time, U_i utilisation of task τ_i. U is the total utilisation of all the tasks in the task-set. α_i is the maximum abort cost for τ_i (see equation 1). $\tilde{C}_i^n$ is the new value for the WCET of τ_i, the biggest abort cost is picked between τ_i and τ_n (see equation 3). In general we allow $D_i \leq T_i$, although previous work and many of the examples in this paper have $D_i = T_i$.

2.1 The Abort-and-Restart Model

In the Abort-and-Restart (AR) model [9], lower priority tasks are preempted and aborted by releases of higher priority tasks. Once the higher priority tasks have completed, the lower priority tasks are restarted as new.

Table 1. An example task-set.

Task	Period	WCET	Release offset	Priority
τ_1	12	3	3	H
τ_2	15	4	0	L

In Table 1 and Figure 1, τ_2 is released at 0 and executes until time 3, because of the arrival of τ_1, τ_2 is aborted at time 3. τ_1 finishes its job at time 6 and τ_2 is restarted as a new job so the spent time between 0 and 3 is wasted.

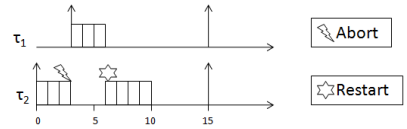


Figure 1. An example task-set.

2.2 Copy-and-Restore Operation

When tasks begin or restart execution, they get a copy (*scratch state*) of the current state from the system [3]. Tasks only modify the copy so no tasks lock the data resource. The copy will be discarded when a higher priority task is released. Once the higher priority tasks have completed execution, the lower priority tasks

are aborted and restarted. When a task has finished, the copy is restored into the system as an atomic action. Although atomic, copy-and-restore cannot be undertaken instantaneously. Hence a high priority task cannot abort a lower priority task while it is restoring state; the higher priority task must block leading to a blocking term in the analysis. For ease of presentation this term is omitted from the scheduling equations given in this paper.

2.3 Related Research

Ras and Cheng [3] state that the critical instant argument from Liu and Layland [7] may not apply fully to the AR model. Table 1 and Figure 1 illustrate this if τ_1 and τ_2 are released together then $R_2 = 7$. Figure 1 shows clearly that $R_2 \geq 10$.

Ras and Cheng [9] also state that standard response time analysis is not applicable for the AR model, and assert that the abort cost can be computed by the following equations:

$$\alpha_i = \sum_{j=i+1}^{N} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot \max_{k=i}^{j-1} C_k \tag{1}$$

$$R_i = C_i + \sum_{\forall j \in hp_i} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j + \alpha_i \tag{2}$$

In Section 3.2 we will derived an equivalent but more intuitive schedulability test for the AR model. Belwal and Cheng [2] noted that Rate Monotonic (RM) priority assignment is not optimal in the AR model. They introduced an alternative policy called *Utilisation Monotonic* (UM) priority assignment in which a higher priority is assigned to a task which has higher utilisation, and showed that it provides better schedulability than RM. They also note that when RM and UM give the same ordering of priorities then that order is optimal (for their analysis).

3 New Analysis

In this section we derive a new sufficient test of schedulability for the AR model. But first we explain why the method cannot be exact.

3.1 Critical Instant for the AR model

First we consider periodic tasks and then sporadic. In the AR model, a critical instant occurs when a higher priority task aborts a lower priority task, because the abort cost is added to the response time. For a 2-task task-set, only the highest priority task aborts the lowest priority task as illustrated in Table 1 and Figure 1. For a 3-task task-set, there are two cases as the highest priority task can abort either of the two lower priority tasks. To generalise:

Lemma 3.1. *A task-set with N periodic tasks under the AR model has at least (N-1)! abort combinations.*

Proof. Consider a pure periodic task-set $\Gamma_N = \{\tau_1, \tau_2, ..., \tau_n\}$ and all tasks only released once. The highest priority task is τ_1 and the lowest priority task is τ_n. Each task τ_i has $N - i$ choices of lower priority tasks to abort. When higher priority tasks are released more than once, the number of choices for those tasks are increased. The number of abort combinations is therefore at least $(N-1) * (N-2) * ... * 1$, which is (N-1)!. □

As there is no information within the task set that would indicate which set of abort combination could give rise to the worst-case response times, they all need to be checked for exact analysis.

For sporadic tasks:

Lemma 3.2. *A sporadic task with a later release may bring a longer response time.*

Proof. In general, a sporadic task with its maximum arrival rate delivers the worst-case response time. Lemma 3.2 can be proved by showing a counter example. In Table 2, there is a three task task-set. Task τ_1 is a sporadic task and has the highest priority. It has a minimum inter-arrival time, 8. Other tasks are periodic tasks.

Table 2. A task-set with a sporadic task.

Task	Period	WCET	Priority
τ_1	8	1	1
τ_2	20	2	2
τ_3	40	4	3

In Figure 2, the response time of τ_3 is 16 when the second job of τ_1 is released with the minimum inter-arrival time, 8.

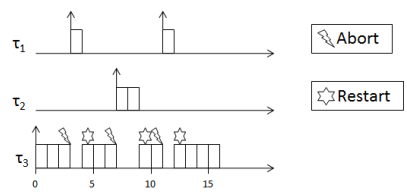


Figure 2. A time chart.

If, however, the second job of τ_1 is released 1 tick later, the response time of τ_3 will be 17. In this case, a sporadic task with a later release may result in a longer response time. □

For a set of sporadic tasks exact analysis would require all possible release times to be checked.

Theorem 3.3. *Finding the critical instant for the AR model with periodic and sporadic tasks is intractable.*

Proof. Lemma 3.1 shows that there is at least $(N-1)!$ abort combinations for N periodic tasks, all of which must be checked for the worst-case to be found. For sporadic tasks all possible release times over a series of releases must be checked to determinate the worst-case impact of the sporadic task. These two properties in isolation and together show that this is an intractable number of release conditions to check in order to define the critical instant. □

A exact schedulability test cannot be tractable if the critical instant cannot be found in polynomial time.

3.2 New Formulation for schedulability tests

In this section we derive a sufficient test that is tractable. Hence we have traded necessity for tractability. We believe this new test is more intuitive than those previously published.

Given a priority assignment, the worst-case response time of task τ_n (priority P_n) will depend only on the behaviour of tasks of priority greater than P_n. Consider the interference caused by a single release of task τ_i ($P_i > P_n$). In the worst-case τ_i will abort, just before it completes, a task with a lower priority than τ_i but with the maximum execution time of all lower priority tasks. Let the aborted task be τ_a, so $P_i > P_a \geq P_n$ and $C_a = \max_{\forall j \in hep_n \bigcap lp_i} C_j$.

The impact of τ_i will therefore be, in the worst-case, C_i at priority P_i and C_a at priority P_a. As $P_a \geq P_n$ this is equivalent (for τ_n) to τ_i having an execution time of $C_i + C_a$ at priority P_i. Let $\tilde{C}_i^n = C_i + C_a$. The original task-set with computation times C_i is

transposed into a task-set with $\tilde{C}_i^n$. This is now a conventional task-set, so the critical instant is when there is a synchronous release. (The maximum interference on τ_n must occur when all higher priority tasks arrive at their maximum rate, initially at the same time, and all have their maximum impact.)

The worst-case for the AR model is that any higher priority task aborts a lower priority task which has the biggest possible worst-case execution time, and that this abort occurs just before the aborted task would actually complete. By this process, a new value $\tilde{C}_j^i$ for τ_j is combined by C_j and C_k:

$$\tilde{C}_j^i = C_j + \max_{\forall k \in hep_i \bigcap lp_j} C_k \tag{3}$$

where $\tilde{C}_j^i$ is the new value for the WCET of τ_j, C_j is the original WCET of τ_j and C_k is the biggest execution time of a task with priority between τ_i and τ_j but τ_j is not included. The response time analysis applies to τ_i. Note that in general the $\tilde{C}_j^i$ values will depend on the task under investigation.

In Table 3, there is an example implicit-deadline task-set. The highest priority is 1. The response time of task τ_4 is being computed.

Table 3. An example with new WCET for 4-task task-set.

Task	Period	C	$\tilde{C}_i^4$	Priority
τ_1	28	2	7(2+5)	1
τ_2	120	3	8(3+5)	2
τ_3	140	4	9(4+5)	3
τ_4	200	5	5(5+0)	4

The $\tilde{C}_i^4$ values are computed by Equation (3). In this example we consider the response time for τ_4 so $i = 4$. For $\tilde{C}_1^4$, j is 1 and C_k is higher than or equal to τ_4 but lower than τ_1. The calculation is $\tilde{C}_1^4 = C_1 + C_4$, so the result of $\tilde{C}_1^4$ is $2 + 5 = 7$.

For $\tilde{C}_4^4$, i and j are 4. C_k is higher than or equal to τ_4 but lower than τ_4 so no task is matched, so the result of $\tilde{C}_4^4$ is $5+0 = 5$. After all the $\tilde{C}_i^4$ values have been calculated, we use $\tilde{C}_i^n$ instead of C in the response time analysis; that is:

$$R_4 = \tilde{C}_4^4 + \sum_{\forall j \in hp_4} \left\lceil \frac{R_4}{T_j} \right\rceil \cdot \tilde{C}_j^4 \tag{4}$$

This is solved in the usual way by forming a recurrence relationship, the result is $R_4 = 36$, which is the same as that obtained via the equation of Ras and Cheng [9]. In fact, the test derived above, i.e. (4), while more intuitive and more efficiently solved is nevertheless equivalent to that given in [9].

Theorem 3.4. *Equations (2) and (4) are equivalent.*

Proof. We rephrase (2) as follows:

$$R_i = C_i + \sum_{\forall j \in hp_i} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j + \sum_{\forall j \in hp_i} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot \max_{k=i}^{j-1} C_k \tag{5}$$

Both $\max_{k=i}^{j-1} C_k$ and $\max_{\forall k \in hep_i \bigcap lp_j} C_k$ pick a bigger WCET task with a priority that is higher or equal to $\tau_(i)$ and lower than τ_j, so we simply to obatin:

$$R_i = C_i + \sum_{\forall j \in hp_i} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot (C_j + \max_{\forall k \in hep_i \bigcap lp_j} C_k) \tag{6}$$

Equation (3) replaces into (6) as follows:

$$R_i = C_i + \sum_{\forall j \in hp_i} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot \tilde{C}_j^n \tag{7}$$

□

As (2) was previously proved to be sufficient for the AR model [9] it follows that (4) is similarly sufficient.

Although the equations are equivalent, (4) is in the standard form for response time analysis and is therefore amenable to the many ways that have been found to efficiently solve this form of analysis [5]. It is also in a form that allows the issue of priority assignment to be addressed.

4 Priority assignment schemes

Here, we introduce a priority assignment policy called Execution-time Monotonic (EM) which assigns a higher priority to a task which has a bigger worst case execution time [1]. An inspection of (3) shows that the minimum execution times (the $\tilde{C}_i^n$ values) are obtained when priority is ordered by execution time. Although this does not necessarily minimise utilisation, it may provide an effective priority assignment policy. Audsley's Algorithm provides optimal priority assignments[1] but it does not hold for the AR model since the response time of a task depends not only on the set of higher priority tasks but also on their relative order (which is not permitted).

4.1 New Algorithm

Exhaustive Search (ES) of all possible priority assignments is optimal for any model but it is not tractable. We used it to validate other policies for small values of N. In a later section, the experiments show that UM and EM have similar results, and they do not dominate each other. If a new algorithm dominates both UM and EM, it will offer a better schedulability rate.

We derive a new algorithm that starts with EM ordering and tests the tasks in priority order starting with the highest priority task. If any task can not be scheduled then try to find a higher priority task which has less utilisation. The ordering begins from the failed task to the top. If a task is found then shift down the higher priority task below the lower priority task. If no task is found, the task-set is deemed to be not schedulable.

Table 4 shows that the task-set is not schedulable at τ_4. Again deadline is equal to period; RT is response time. Note only C values are given in the table, the necessary $\tilde{C}_i^n$ values are dependent on which task is actually being tested, they must be re-computed for each task.

Table 4. An example task-set fails in EM ordering.

Task	Period	C	U	Priority	RT
τ_1	60	6	0.1	1	6
τ_2	50	5	0.1	2	16
τ_3	32	4	0.125	3	24
τ_4	25	3	0.12	4	30 (X)
τ_5	100	2	0.02	5	

τ_2 has less utilisation than τ_4 so we shift τ_2 down below τ_4. The new ordering is τ_1,τ_3,τ_4,τ_2 and τ_5, then the task-set is schedulable. We call this policy Execution-time-toward-Utilisation Monotonic (EUM) priority assignment.

[1] With ties broken arbitrarily

4.2 Time complexity

To analyse the complexity of the EUM policy, we count each single task schedulability test required (each test is itself of pseudo-polynomial complexity). In the worst-case, an N-task task-set starts with EM ordering and the task-set is only scheduled by UM ordering which is the completely opposite to EM. It is easy to see that in this case, $2N - 1$ schedulability tests are required before the task that starts out at priority N is placed at priority 1, and that a further $2(N-1) - 1$ tests are needed before the next task (that started at priority $N - 1$) is placed at priority 2. Overall, the number of single task schedulability tests required to transform EM ordering into UM ordering is given by:

$$\sum_{k=1}^{N-1} (2k - 1) = N^2 \tag{8}$$

So the complexity of EUM priority assignment is $O(N^2)$ single task schedulability tests. EUM dominates EM and UM because the EUM algorithm starts with EM ordering and ends at UM ordering in the worst-case; however, unlike Exhaustive Search (ES) it is a tractable priority assignment policy.

5 Experimental Evaluation

The experiments compare different priority assignments (DM, UM, EM, ES and EUM) for the AR model. The parameters are: Deadline is equal to period. All tasks are periodic. A set of N utilisation values U_i was generated by the UUniFast Algorithm [4]. Task periods were generated between 500 and 5000 according to a log-uniform distribution[2]. Task execution times are: $C_i = U_i \cdot T_i$ Utilisation for task-sets are ranged between 20% and 60% in steps of 1%. 10000 task-sets were generated for each utilisation level. The number of tasks in each task-set was 8, as this is the maximum that could be handled by Exhaustive Search (ES). Other experiments were also performed for larger task sets (up to 20 tasks) for the heuristic policies, but are not shown due to space limitations.

In Figure 3 the X-axis is Utilisation and the Y-axis is the Schedulability rate, i.e. the percentage of task sets that were deemed schedulable. DM has the worst schedulability, UM and EM are quite similar, while EUM is the best of the heuristics and very close to optimal for task sets of size 8. Indeed it is impossible to distinguish between them. Nevertheless EUM is not optimal, the figure contains in total 410,000 task sets of which ES deemed 137,366 schedulable and EUM (136,712), a difference of 654 (i.e. schedulable by ES but not by EUM). Note that EUM explored a maximum of $N(N-1)/2 = 28$ different priority orderings in this case, whereas ES explored a maximum of $N! = 40320$. Although not exact, the performance of EUM for $N = 8$ leads to a reasonable conclusion that EUM is an effective and near optimal priority ordering for the AR model, at least for relatively small task sets.

6 Conclusion

The AR model has been proposed as a means of implementing priority-based functional reactive programming. Any released task, if it has a higher priority than the current running task, will abort that task. It can therefore immediately make progress. As a consequence the aborted task must re-start its execution when it is next executed.

[2]The log-uniform distribution of a variable x is such that ln(x) has a uniform distribution.

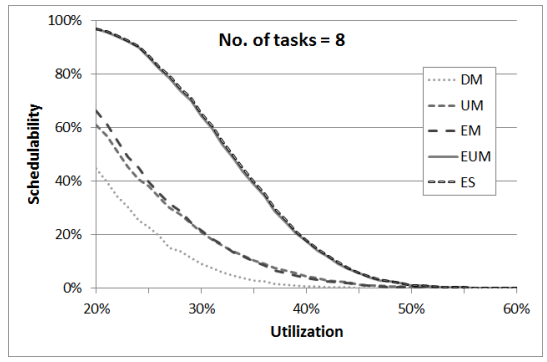


Figure 3. The number of tasks is 8.

We have confirmed that the AR model is intractable, in the sense that exact analysis is not possible due to the number of cases that need to be investigated in order to identify the worst-case release conditions (the critical instant). Nevertheless a tractable sufficient test has been developed that allows the issue of priority ordering to be addressed.

Unfortunately optimal priority ordering is also problematic with the AR model. Deadline (or Rate) monotonic ordering is demonstrably not optimal. Also the optimal Audsley's algorithm is not applicable. We have however developed a heuristic (called EUM) that performs well and has $O(N^2)$ complexity in terms of the number of single task schedulability tests required. On small sized systems ($N = 8$) EUM performs almost identically to an optimal scheme (using exhaustive search). For larger numbers of N (where exhaustive search is infeasible) it performs better than previous published approaches.

References

[1] N.C. Audsley. On Priority Assignment in Fixed Priority Scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

[2] C. Belwal and A.M.K. Cheng. On Priority Assignment in P-FRP. *RTAS*, pages 45–48, 2010.

[3] C. Belwal and A.M.K. Cheng. Determining Actual Response Time in P-FRP. In Ricardo Rocha and John Launchbury, editors, *Practical Aspects of Declarative Languages*, volume 6539 of *Lecture Notes in Computer Science*, pages 250–264. Springer Berlin/Heidelberg, 2011.

[4] E. Bini and G. Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30:129–154, 2005.

[5] R.I. Davis, A. Zabos, and A. Burns. Efficient Exact Schedulability Tests for Fixed Priority Real-Time Systems. *IEEE Transactions on Computers*, 57(9):1261–1276, 2008.

[6] R. Kaiabachev, W. Taha, and A. Zhu. E-FRP with priorities. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 221–230. ACM, 2007.

[7] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, 1973.

[8] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek. Preemptible atomic regions for real-time Java. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 10 pp.–71, 2005.

[9] J. Ras and A.M.K Cheng. Response Time Analysis for the Abort-and-Restart Task Handlers of the Priority-Based Functional Reactive Programming (P-FRP) Paradigm. In *Embedded and Real-Time Computing Systems and Applications, RTCSA*, pages 305–314, 2009.

[10] H. Takada and K. Sakamura. Real-time synchronization protocols with abortable critical sections. In *Proc. of the First International Workshop on Real-Time Computing Systems and Applications*, pages 44–52, 1994.

[11] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *Proc. of the ACM SIGPLAN*, PLDI, pages 242–252. ACM, 2000.

Load and Quality Cooperation for Distributed Embedded Systems Using Different Modes of Operation

John F. Schommer
Embedded Software Laboratory
Ahornstr. 55
52074 Aachen, Germany
schommer@embedded.rwth-aachen.de

Thomas Gerlitz
Embedded Software Laboratory
Ahornstr. 55
52074 Aachen, Germany
gerlitz@embedded.rwth-aachen.de

Stefan Kowalewski
Embedded Software Laboratory
Ahornstr. 55
52074 Aachen, Germany
kowalewski@embedded.rwth-aachen.de

ABSTRACT

Nowadays the design goal of embedded systems is shifted from statically defined to a dynamic quality of service at application level. Where embedded systems getting distributed or co-located on platforms, the complexity increases along with their adaptivity and dynamic. And while real-time requirements are still essential, in that setup the runtime behavior gets unpredictable at compile time leading to this trend. This paper presents a software-architecture for embedded systems connected within networks to reach a common computational objective. This architecture enables the network to instantaneously switch its mode of operation. The mode switch is done in two steps, coordination, where a dynamic event is consumed, and adaptive transitions, where the mode switching is delegated to all network nodes. Finally, the paper presents one proof of concept and ongoing as well as related work.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-based Systems**]: Real-time and Embedded Systems; C.4 [**Performance of Systems**]: Fault Tolerance

Keywords

Load balancing, Quality cooperation, Modes of operation

1. INTRODUCTION

Nowadays embedded systems are getting distributed or co-located on platforms. Although real-time requirements are still essential in such networks, this trend forces distributed embedded systems to become more adaptive and dynamic, where the complexity increases as well. While running a couple of applications simultaneously within one network or on one platform, the runtime behavior gets unpredictable at compile time. The design goal for distributed embedded systems is therefore shifted from a statically defined setup to a dynamic quality of service at application level. In the worst case, the network shall keep performing its main function in the presence of faults, when several resources are temporarily or even permanently not available.

For instance, a next generation electric car might be fully controlled by drive-by-wire. This car will have a distributed power and wheel controlling architecture, where four wheels are driven by four electric motors controlled by four controllers. Although the design goal is robustness, the developers must consider fault tolerance on the level of safety-critical systems.

Instead of using for instance *Triple Modular Redundancy* [5], with fixed redundant hard- and/or software components, different modes of operation can be designed taking into account that some resources may not be available at runtime. If for example one wheel controller of the next generation car fails, the processing must be either immediately redistributed over the three still functional controllers, or the whole network switches to a degrade mode of operation with less computational consumption.

This paper presents an architecture to enable dynamic quality of services for distributed software of embedded systems using different modes of operation. The described architectures extends the concept of *graceful degradation* [4, 1] to a more flexible approach, where adaptive transition allows *quality degradation* and also *computational promotion*. This leads to a network of embedded systems that can cooperate with respect to load and quality. The remainder of this paper is structured like follows. In Section 2 the problem is discussed in more detail. After the contribution of this work is summarized in the next section, a software-architecture is presented in Section 4 that can be applied to the specific problem. After that one feasibility study is presented, followed by discussions about related and ongoing work. Section 7 concludes this paper.

2. LOAD AND QUALITY BALANCING

During the runtime of distributed embedded systems various events might occur that interrupt the typically periodic control flow of the software. Reconsider the example stated in Section 1. One wheel controller of the next generation electric car causes a fault. In current generation cars this fault may lead to graceful degradation [4, 1], which is a well known concept for safety-critical embedded systems. In this mode of operation the car would stop controlled and safely, hopefully without harming any occupants. Of course, this concept can also be applied to the design of the next generation car.

But the presented problem description goes further, considering the distributed embedded systems as network entities cooperating for a common objective. If either, depending on a dynamic event, more resources are needed or existing ones are no longer available, the remaining network nodes have to adapt in a way that this common objective can still be reached. For the sketched scenario, this paper will use the term cooperative network like defined.

Definition 1: Cooperative networks.

A cooperative network is a set C of distributed embedded systems that are physically or logically connected via some kind of network topology T.

In the next generation car example the common objective is that

the car stays functional. Even if one of the wheel controller is failing, the car can adapt in two ways to reach this objective; only powered by the three remaining motors respectively wheel controllers. The first way is to increase the computation consumption, so that the result of the computation stays constant in terms of quality. Following the example, this means that the three remaining wheel controllers must, along with their default computation, compute the output of the failing fourth controller, where the quality of the computation stays on the default level. For simplicity, it is assumed that in this example the results of these computations are broadcasted via a bus system connecting each component of the systems. The second way is to decrease the quality of the computation result, so that the computation consumption stays constant in terms of energy. Again, the three remaining wheel controllers compute the output of the failing controller. But now, the quality of the computation is degraded to a level, where the energy consumption of the wheel controllers stays constant. In the given example the mode switching can be done by reducing the frequency of calculation iterations within each wheel controller. It is additionally required that a cooperative network can recover to a default mode of operation, if this mode is again suitable depending on the concrete scenario, e.g. if the defect node successfully resets.

3. CONTRIBUTION

This paper targets distributed embedded systems that need to coordinate together to reach a common objective. An architectural framework is sketched to establish different modes of operation as quality of service at application level for distributed embedded systems. Two strategies are presented dealing with load redistribution among the nodes and the quality degradation of the functional behavior of the cooperative network. This framework provides methods to systematically cope with the problem description presented above including that the network can return to the default mode of operation, if possible. Hence, an overview of challenges in hindsight of the coordination of the nodes when deployed in different topologies is given.

4. CONCEPT

This section describes the approach of how to realize the requirements sketched in Section 2. To make the presented approach easier to illustrate, Figure 1 abstracts from the electric car example to the computation of geometric figures. These geometric figures are used for illustration in the remainder of this paper. In both examples three modes of operation are sufficient: default, increased consumption and decreased quality.

In this cooperative network the network nodes are working together to compute the outer edges of a square, where the common objective of the network is to compute a closed polygon. The computation is implemented in four tasks and distributed on four nodes, e.g. to four different dual-core processors or using Ethernet. In default mode, the work load is equally distributed to each node by computing one of the edges. When now $Node_1$ fails, two options exist, similar to the first example, to cope with this failure. Increase the workload of the remaining nodes, which for example means to compute one and one third edges, or degrade the functional behavior of the whole system, which for example means to compute a triangle, which is also a closed polygon, instead of a square.

Depending on the application scenario one of these two strategies can be applied. To accomplish this application, the presented approach uses the term adaptive transition like defined.

Definition 2: Adaptive Transition.

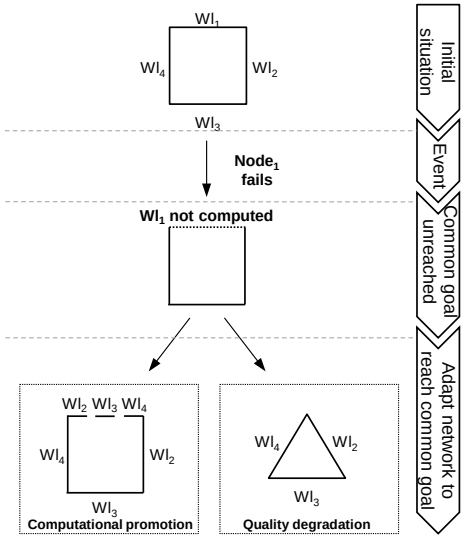


Figure 1: Problem Illustration

A cooperative network C can switch from one mode of operation to another mode with one of the following adaptive transitions.

- *Computational Promotion:* Increase the computation consumption of a subset of C, to keep the quality of the computed result constant. *Load* is a common synonym to computation consumption.
- *Quality Degradation:* Degrade the quality of the computed result to keep the computation consumption constant.

A cooperative network is enabled to makes adaptive transitions, exactly if:

1. A coordinator node exists within the network. Any dynamic event occurring in scope of the network is handled by this coordinator node.
2. Each node in the cooperative network implements an interface, where the coordinator node can force switching modes of operation.
3. A unique behavior table is implemented by each node in the cooperative network. When the coordinator node declares another mode of operation, each node instantiates this new mode of operation depending on this unique behavior table.

How many transitions are allowed in one direction is only limited by the implementation, e.g. the amount of modes of operation that may be instantiated. Due to page limitations, the remainder of this paper assumes that a load increase directly leads to maintaining quality.

4.1 Coordination

Distributed embedded systems are typically organized in a network topology like the star, ring, bus or full mesh topology, but may be organized within other common topologies. Together with a corresponding type of communication the topology has a major impact on how the cooperation and the adaptive transition have to be implemented. Figure 2 shows a possible overall architecture as an OSI-Model. The proposed architecture resides on the application layer of the respective nodes and depends on the scheduling

and networking services of the used operating system. Highly reactive scheduling in case of failures, prompt network transmissions and a reliable physical layer are needed for the coordination process. The core components of the proposed framework include a *coordinator node* responsible for the coordination of the computation of a common objective within the cooperative network. This computation is executed by so called *ordinary nodes*, which provide the interface to the coordinator node to interact with. This interface is used to switch the mode of operation of the ordinary nodes via a *behavior table*.

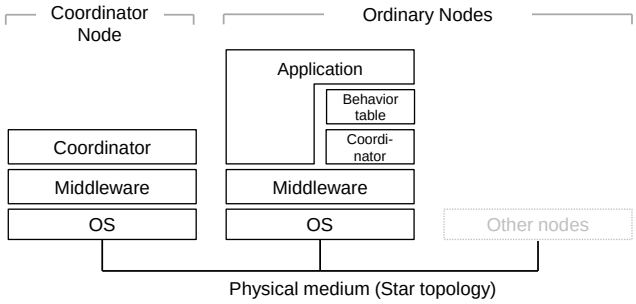


Figure 2: Cooperative Architecture

To present the concept of how network nodes can coordinate together in a cooperative network when an event occurs, this paper focuses on networks of embedded systems with star topologies. The central node in such a network is considered to be the coordinator node.

Definition 3: Coordinator Node.
A coordinator node is a node within the cooperative network that is responsible for switching the modes of operation within the network.

Considering the illustrating example, such a coordinator node coordinates the computation of the outer edges of the square. When now one of the four network nodes computing single edges fails, the coordinator must

1. receive this event and
2. accordingly adapt the rest of the network to another mode of operation.

Receiving an event, which means in the example to determine a node failure within the cooperative network, will be accomplished as follows: if only a subcomponent of one network node fails, in the example e.g. one of two cores of a processor used to compute outer edges, and if the node is still able to communicate, this event is directly sent to the coordinator node. This coordinator can then switch the mode of operation within the network, which means that all still functional nodes in the network are forced to switch their mode of operation instantaneously. If the whole node is not available anymore, e.g the software of the node faces a failure, the node is rendered unable to communicate with the coordinator node. In this case a watch dog design pattern, which can be implemented within the coordinator and/or other ordinary nodes, can be used. The locations of the watch dogs within the cooperative network highly depends on the used network topology, i.e. when ordinary nodes can only communicate with the coordinator node (star topology) they do not need to implement a watch dog for other ordinary nodes. After detecting such an event the network switches to another mode of operation like in the previous case.

While ordinary nodes may fail within a cooperative network, a failure of the coordinator renders the system unadaptable in case of further failures within the system. Recovering from a failure of the coordinator node is only rational, if all remaining ordinary nodes can still communicate with each other and a new coordinator node can be elected. This can be accomplished by an algorithm which chooses the new coordinator node on the basis of the available resources of a node. Similar approaches are used in coordination mechanisms for wireless sensor networks.

4.2 Topological Issues

When considering these two event-based approaches for a star topology, they can be ported on logical level to all other topologies. But on the other hand, applying them might cause a severe communication overhead. Therefore, different approaches might be better suited for these topologies. For example, self-organized coordination, as already applied in the field of sensor networks, might be a desirable property of the mesh topology. These enhanced communication paradigms will be discussed in ongoing work based on this paper.

4.3 Instantiating modes of operation

After the coordinator has forced the still functional network nodes to switch their mode of operation, these nodes must perform an local adaptive transition. Depending on the resulting computation this transition leads to a computation promotion, which implies a workload redistribution throughout the network, or a quality degradation, which is similar to the concept of graceful (functional) degradation of the network. Comparing adaptive transitions like defined in this paper with graceful degradation, it is not a fundamental new approach but provides a more flexible way of event-based reaction. Within the presented architecture the modes of operation are coded into a behavior table at design time. Each single node in the cooperative network must implement such a behavior table like defined.

Definition 4: Behavior Table.
A two-dimensional matrix, where the rows define the available modes of operation and the columns define the workload of the implementing network node, when the specific mode of operation must be executed. The modes of operation are delegated by the coordinator node to the ordinary nodes, where the column definition allows a parametrized mode switching. The entries of behavior tables define a set of algorithms, e.g. function pointers in the Programming Language C.

The set of algorithms should be fully calculated at compile time or must be dynamically safeguarded to ensure feasible transitions only, which is typically an NP-hard problem.

Depending on such a behavior table, the nodes of a cooperative network determine their local adaptive transition. After the transitions are executed for all available network nodes, the cooperative network has switched its mode of operation. For example if one of two cores of the processor of $Node_1$ fails, the behavior tables of the remaining nodes can be implemented like follows:

- The algorithm to compute the outer edge of the square is simply executed on the remaining core of $Node_1$.
- $Node_2$, $Node_3$ and $Node_4$ just generate default output.

5. PROOF OF CONCEPT

The presented architectural approach has been applied in an application scenario, where a simple fan has been constructed, powered by three electric motors. The architecture provides a workload redistribution mechanism to this setup.

The fan is powered by three motors combining their output via two differentials into one axis. These motor controllers are coordinated by one microcontroller unit, with limited computational power (ARM processor AT91SAM7S256 running at 48 MHz and 64 kB RAM), running three controller tasks and a centralized coordinator (star topology) in quasi-parallel manner. The coordinator task detects failures as discussed in Section 3, by applying the concept of computational promotion. Therefore a motor failure does not result in the lock of the whole fan system. If one of the motor controllers fails, the rotation of the fan is kept at a certain speed. This is achieved by increasing the load of the remaining motors and therefore promoting the state of the remaining motor controller.

The presented architecture also enables the fan system to revert to the default state, if a failed controller has recovered. The resulting rounds per minute (RPM) of the fan system are measured using a incremental rotary encoder. The evaluation of the results show that the RPM of the fan system are not significantly affected in the presence of failures or during both transitions to the adaptive mode of operation or afterwards back to the default mode.

6. RELATED WORK

Graceful Degradation is a important design pattern in safety-critical systems [4, 1], in particular for fault-tolerant systems. The presented work goes beyond this statical approach, providing a flexible concept to recover from the computation of dynamic events and introduces the concept of computational promotion beside the already applied concepts of performance degradation [6].

A optimized graceful degradation is presented in [2]. The optimization is done on structural and behavioral level, and here in a decentralized fashion. But, unlike the approach in the present paper optimized graceful degradation does not consider the energy consumption and the ability to recover from degradation.

Another similar approach is presented in [3], where tasks are reconfigured by moving them in a network of connected devices. This work makes use of reconfigurable hardware like programmable logic controllers to adapt to failed devices and newly introduced tasks in an online fashion. On contrast to this work, the presented approach focuses on software, which also allows not-reconfigurable hardware that are getting more and more important in current distributed embedded systems, but comes with the constraint to make the behavior table safe.

7. CONCLUSION

In this paper a software architecture for distributed embedded systems is presented that allows cooperative work to reach a common objective in the presence of dynamic events and even failures. Load and quality of the application running on the network is balanced by using instantaneously switching modes of operation in case these events occur. The design pattern presented in this paper can be deployed to safety-critical software with respect to the underlying software platform of the network nodes. The architecture is evaluated based on a network of nodes arranged in a star topology on the physical layer, which is introducing a single-point-of-failure regarding the coordinating node.

The presented concept is work in progress. Therefore, this single-point-of-failure and other challenges like scalability are in detail tackled in ongoing work. One notable advantage of the presented approach is the opportunity to allow also computational promotion of the network using additional modes of operation. But due to page limitations, such scenarios are not discussed in the present paper. Another advantage of this approach is the ability to recover from dynamic extraordinary events via the presented concept back to a default state. Considering that this approach is fully software-oriented, this is more or less an advantage for software-based developing safety-critical applications. The presented approach is limited by the timing behavior of the underlying platform. It has to be evaluated, how good the concepts for load and quality balancing work in terms of scalability and protocol stack dependencies. This involves other topologies of networked embedded systems as well as nested adaptations of parts of the cooperative network. Further it must be analyzed, if the architecture can be used in priority-based network communication. In this context, overhead of data flow and communication between the nodes of the network requires careful planning and evaluation.

The presented approach cannot eliminate the influence of bad programming of the design pattern. but as a next generation electric car in model size is available, it is planned that the presented work is evaluated exhaustively and during a larger case study under more realistic parameters. Conclusions are expected from this feasibility study regarding of how to force good programming also.

8. ACKNOWLEDGMENT

This work was supported by the UMIC Research Centre, RWTH Aachen University Germany.

9. REFERENCES

[1] A. Avizienis. Fault-tolerance: The survival attribute of digital systems. *Proceedings of the IEEE*, 66(10):1109–1125, 1978.

[2] M. Glass, M. Lukasiewycz, C. Haubelt, and J. Teich. Incorporating graceful degradation into embedded system design. In *DATE*, pages 320–323. IEEE, 2009.

[3] C. Haubelt, D. Koch, F. Reimann, T. Streichert, and J. Teich. Reconets design methodology for embedded systems consisting of small networks of reconfigurable nodes and connections. In M. Platzner, J. Teich, and N. Wehn, editors, *Dynamically Reconfigurable Systems*, pages 223–243. Springer Netherlands, 2010.

[4] T. Henzinger and J. Sifakis. The embedded systems design challenge. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2006.

[5] R. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[6] K. Shin and C. Meissner. Adaptation and graceful degradation of control system performance by task reallocation and period adjustment. In *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, pages 29–36, 1999.

Design and Implementation of a FPGA-Based RTOS Real-Time Performance Analysis Environment (RTPE) for Satellite On-Board Computers

Fernando Nicodemos, Osamu Saotome
Instituto Tecnológico de Aeronáutica - ITA
Pç. Marechal Eduardo Gomes, 50 - IEEA
São José dos Campos - SP - Brazil
fgnicodemos@gmail.com, osaotome@ita.br

George Lima
Universidade Federal da Bahia - UFBA
Av. Adhemar de Barros, s/n - DCC
Salvador - BA - Brazil
gmlima@ufba.br

Abstract

We address the problem of measuring context switching overhead in Real-Time Operating Systems (RTOS). Such characteristic lies in the core of RTOS performance and is usually assessed via custom vendor software routines or benchmarking. The former approach can be too intrusive while the later may incur in tendentious measurements. In this paper we describe an external non-intrusive FPGA-based measurement tool. Our approach has been applied to a space case study for which the Real-Time Executive for Multiprocessor Systems (RTEMS) RTOS and the ERC32 processor are used. Experiments evaluate the effectiveness of our tool.

1. Introduction

The complexity of space embedded systems has increased substantially in the past few years due to new missions that demand higher on-board processing capabilities. Indeed, silicon foundry has evolved, new architectural organizations has led to higher performance, and applications, once limited to lower level languages (*e.g.* assembly), are now developed in higher level languages (*e.g.* C language). Combining these aspects with mission costs and code reuse needs for different projects, the use of a RTOS for supporting space embedded systems is almost inevitable. However, when implementing an embedded hard real-time system, one has to account for the effects that the RTOS has on the application. Task context switching, for instance, have direct impact on the application timeliness. Thus, the main requirement for a RTOS-based space application is to prove that the real-time behavior is predictable, *i.e.* that all tasks complete before their deadlines in the worst-case conditions [1].

Usually, measuring RTOS core characteristics is carried out by software-related approaches, via a timing test suite, *e.g.* the user extensions of RTEMS Timing Test Suite [2][3]. Benchmarking, like Rhealstone [4], Hartstone [5], MiBench [6] and Thread-Metric [7], has been also used. Static analysis has been also used to make hard real-time systems comply with space standards [8][9][10][11]. These approaches offer some drawbacks since no standard measurement methodologies exist and overestimation is inevitable.

Thus, hardware tools for supporting such RTOS core analysis and carrying out such measurements are thus justifiable and of paramount importance. Emulation of an embedded system for industrial automation has been considered in [12]. The goal is to know the conditions and architectural combinations of a proprietary hardware operation aiming to the best overall application gain while reducing design time and risks. In [13] a specialized hardware and software module has been described. The module was created to generate control signals over a RS232 interface so that a spy system can calculate event response times. RTOS core analysis was not taken into account. The Lauterbach company have presented the TRACE32 debugger. Data can be statistically evaluated and graphically displayed [14]. This solution depends on the PowerPC architecture and supports specific RTOS versions, *e.g.* the RTEMS 4.6.x.

An initial version of a FPGA-based measurement tool has been reported [15]. In this paper we describe a further approach of this tool named Real-Time Performance Analysis Environment (RTPE). It is used for assessing RTOS core performance without being too intrusive and capable of giving precise measurement results. The RTPE is composed of three components: a host computer (HOST PC), a FPGA Measurement Unit (FMU) and Device Under Test (DUT). We have applied the RTPE for analysing RTEMS. In the following sections we detail this approach and present experimental results. Although we have successfully applied the described approach to analyzing other RTEMS characteristics, such as interrupt, we focus here only on the context switching analysis.

2. Context switching with preemption

Context switching is an operation that occurs when one task leaves its own context and another task takes

over the processor with another context. Since RTOS does not have information about the current register scheme, the whole register set is saved for the current context. The new context is restored, the processor pipeline is flushed and specific RTEMS-related routines are executed. The total time to execute the operation described above is called context switching timing or overhead. One of the main difficulties in assessing this overhead is due to the low-level operations involved. Thus, instrumenting the RTOS can be too intrusive. Our context switching measurement model can be executed in RTEMS at user application code without modifying any RTOS code. We use special purpose tasks to generate signals so as to allow that an external measurement tool is able to measure context-switch time. A minimum of two tasks are needed to provide context-switch events. As our FPGA-based tool has four input pins, we do not considered more than four tasks in our experiments. Similar results would be obtained if less than four tasks were considered. Moreover, we note that RTEMS is compiled only with the non-floating point library and therefore context switching that includes floating point libraries are not evaluated.

We have considered the round-robin scheduling model of RTEMS. Every task joins a FIFO queue and each task has equal and configurable timeslices without any weight and priority when executing. Figure 1 illustrates four monitored tasks being scheduled according to the round-robin policy. That is, after a task executes its timeslice, the scheduler selects the next one on circular queue.

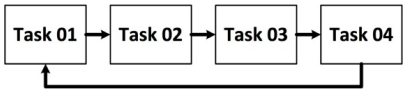


Figure 1. Context switching model with preemption.

A GPIO pin from the DUT is assigned to each of these tasks. The corresponding GPIO is set to mark logic level "1" and cleared to mark logic level "0" within task application code, creating a squared waveform. The GPIO transitions allow the FPGA-based to measure the absolute time differences and to define when one task reaches the total timeslice execution and when the next task starts executing. Figure 2 illustrates their temporal behaviour.

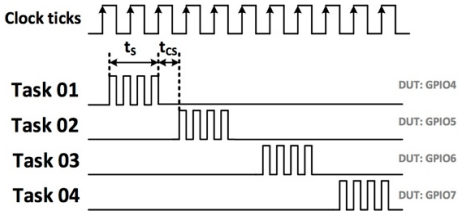


Figure 2. Context switching timing chart.

A task runs its timeslice (t_S), configurable by setting the RTEMS "microseconds per tick" variable, and then the scheduler selects the next task to run. The context switching time can then be measured (t_{CS}). Clock ticks are set by adjusting RTEMS "ticks per timeslice" granularity variable.

3. Space case study

The ERC32 processor was chosen, following the roadmap by the Brazilian space program. It is a high-performance 32-bit RISC embedded processor, implementing the SPARC architecture V7 specification. The TSC695E Starter Kit from Atmel [14] was used for running the RTEMS core analysis proposed in this work. The kit uses the low voltage version with part number TSC695FL-15MA-E (ERC32 single chip), ordered as an engineering sample. It operates at a maximum core clock of 30 MHz [16]. Actually, the ERC32 core clock is set at 12 MHz, driven by an external oscillator operating at 24 MHz (core clock is automatically divided per two).

RTEMS is also defined following the roadmap by the Brazilian space program. It is a free open-source real-time executive designed for embedded systems characterized by three layers: hardware support, kernel and APIs. The hardware support layer encompasses the processor and board dependent files as well as a common hardware library. The kernel layer is the heart of RTEMS and encompasses several libraries. The API layer makes the bridge between the kernel and the user application [17]. The analysis of this work ran on the current RTEMS version 4.10.2.

4. RTPE Environment

Several approaches for measuring RTOS core characteristics exist and were presented in Section 1. Although hardware-based measurements may offer accurate data, carrying out such an approach involves specialized knowledge about both the hardware and the software architecture to be assessed. To avoid the need for this type of knowledge, we have developed the RTPE tool to assess RTOS core characteristics, under contract with the Brazilian Space Agency (AEB), via the UniSpace Program.

RTPE is based on an external FPGA Measurement Unit that can automatically read and store time transitions to generate real timing information for the model described in Section 2. The counterpart of the model was also developed and implemented into the FPGA (a mix of megafunctions and VHDL language were used). The RTPE environment is composed of three basic functional blocks: 1) Host computer (HOST PC); 2) FPGA Measurement Unit (FMU); and 3) Device Under Test (DUT). Figure 3 shows the three RTPE functional blocks applied to the case study.

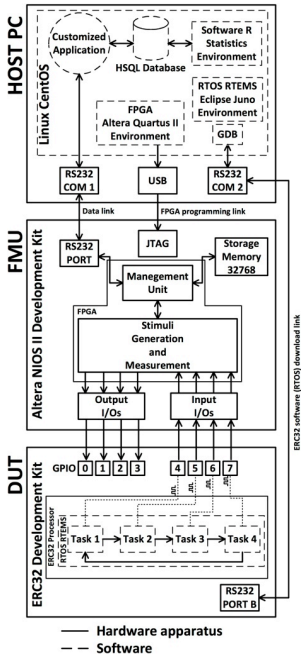


Figure 3. Real-Time Performance Analysis Environment - RTPE.

The function of the HOST PC is to provide the software development environment for all the subcomponents of the other two blocks. It uses the Linux CentOS 6.2 as the host OS. The RTEMS application code based on the model described was developed using the classic API, using the Eclipse Juno Environment. The FPGA development environment is the Altera Quartus II 7.2 [18]. All other software used to design and implement the RTPE is focused on open-source or free solutions. We developed a custom application in Java to make the bridge between the collected data from FMU with the open-source HSQL database [19]. The post analysis is conducted via the R environment [20].

The function of the FMU is to provide the non-intrusive tool for stimuli generation and timing measurements of the model presented, under operation in the DUT. It acts as a digital 50 MHz sampler so that the time transitions can be stamped with a 32-bit counter. The absolute differences between the time stamped in these transitions are identified and time events due to tasks transitions can be measured. The FMU stores a minimum of three time transition samples to identify one context-switch overhead between tasks. This block makes use of the Altera NIOS II Development Kit [21].

The DUT is the device that one shall test under operation in which the RTEMS core characteristic will be measured.

RTPE has a simple operating principle. First, the RTEMS application code, corresponding to the context switching model, is downloaded via a RS232 interface from HOST PC to DUT. The FPGA module is programmed with its counterpart model via the JTAG interface. The pressing of a button in FMU enables the measurement and the first transition in any GPIO is timestamped as 0. The following transition times are timestamped accordingly. Up to 32,768 5-byte transition time samples can be locally stored in the FMU. One byte carries the task-related GPIO information and four bytes carry the 32-bit timestamp values associated with each sample. These values correspond to the absolute timing differences read from the GPIOs 4-7, signaled from DUT. The measurement procedure ends when memory is full, which is automatically detected by the FMU. After this, the stored data can then be sent to the HOST PC via the RS232 interface using a custom Java application. Collected data is then analysed.

5. Results

Three data sets of 32,768 transition samples were collected via RTPE tool for the context switching model. Task timeslice values were considered to be 1, 10, and 100 ms, each value for a collected set. Figure 4 shows histograms for the calculated context-switch overheads for each set.

The histograms show that the higher the timeslice considered, the higher the occurrences near the worst border. It also shows a higher variability, resulting from the hardware architecture and the RTEMS context-switch software proceedings described in Section 2. However, it is worth of notice that the interval between the best and the worst-case observed values lie within 130-160 microseconds independently of the timeslice used. In Table 1, the observed best and worst times for the three data sets are given.

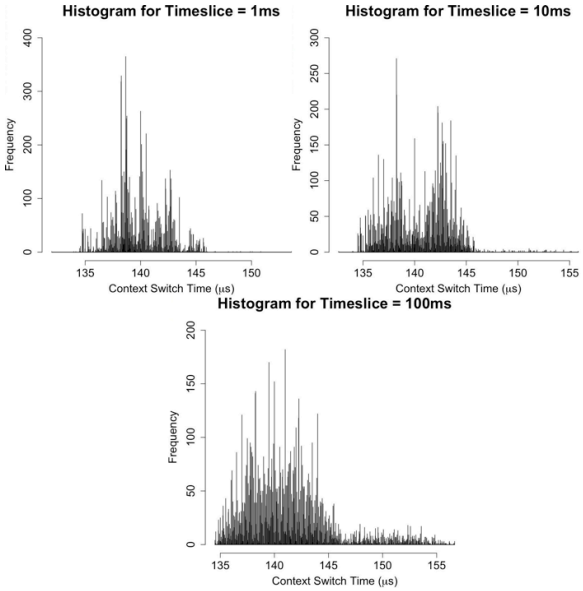


Figure 4. Context switch data histograms.

Table 1. Context switching measured timing overheads.

Context Switching Model with Preemption			
Timeslice / Ticks per ts	***Best Case***	***Worst-case***	***Mean / Std. deviation***
1 ms / 1	131.98 µs	153.6 µs	139.7898 µs / 2.312548
10 ms / 1	132.64 µs	155.84 µs	140.4472 µs / 2.975035
100 ms / 1	134.5 µs	156.66 µs	141.2638 µs / 4.024658

6. Conclusions

The Real-Time Performance Environment (RTPE) environment was presented as a hardware-based non-intrusive time measurement approach. We have analyzed RTEMS in terms of its overhead when dealing with a context switching model. Observed measurements indicated that most context-switch overhead lies in between 130-160 microseconds for the considered hardware platform. The observed tight distribution for the sampled data indicates certain stability of RTEMS, a characteristic required for space applications such as satellite on-board computer.

The RTPE tool adds indeed a new way to assess RTEMS core performance and can complement other methods to comply with space standard needs. Future work includes the development of new RTEMS core analysis models and its counterpart implementation in RTPE. Floating point RTEMS library compilation should be studied and also analyzed with the models. The impact of cache memory should also be included. Another aspect that can be considered is the impact of the Error Detection And Correction (EDAC) unit available in ERC32. Indeed, the detection and correction of possible bit errors due to transient faults is an important parameter to be analyzed. Further, a comprehensive statistical analysis should also be conducted with the measured data so that RTEMS developers can identify software and optimize routines.

References

[1] "ECSS-E-ST-40C: Software", European Cooperation for Space Standardization, 03/06/2009.

[2] On-Line Application Research Corporation, “RTEMS C User's Guide”, www.rtems.com, 2013.

[3] On-Line Application Research Corporation, “RTEMS Intel i386 Application Supplement”, www.rtems.com, 2003.

[4] R. P. Kar and K. Porter, “Rhealstone - A real-time benchmarking proposal”, Dr. Dobb's Journal, vol. 14, pp. 14-24, 1989.

[5] W. A. Halang, R. Gumzej, M. Colnaric and M. Druzovec, “Measuring the performance of real-time systems”, International Journal of Time-Critical Computing Systems, Vol.18, pp. 59-68, 2000.

[6] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite", Proceedings of the Workload Characterization, pp. 3-14, 2001.

[7] W. Lamie and J. Carbone, “Measure your RTOSs real-time performance”, www.eetindia.com, 2007.

[8] C. Brandolese, and W. Fornaciari, "Measurement, analysis and modeling of RTOS system calls timing", 11th Euromicro Conference on Digital System Design, pp. 618-625, 2008.

[9] A. Colin, and I. Puaut, "Worst-case execution tome analysis of the RTEMS real-time operating system", 13th Euromicro Conference on Real-Time Systems, pp. 191-198, 2001.

[10] M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu and J Zhang, "A survey of WCET analysis of real-time operating sustems", International Conference on Embedded Software and Systems, pp. 65-72, 2009.

[11] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat and P. Stenstrom, "The wors-case execution-time problem: overview of methods and survey of tools", Journal of ACM Transactions on Embedded Computing Systems, Vol.7, Issue 3, Article 36, 2008.

[12] K. Weiβ, T. Steckstor and W. Rosenstiel, “Performance Analysis of a RTOS by Emulation of an Embedded System”, 10th IEEE International Workshop on Rapid System Prototyping, pp. 146-151, 1999.

[13] I. V. Druzhinin, A. A. Mynzhasova and E. A. Sinelnikov, “Design and implementation of a hardware-software module for testing real-time systems”, 34th International Convention MIPRO, pp. 788-791, 2011.

[15] http://www.lauterbach.com/frames.html?home.html

[15] L. H. Shibuya, S. S. Sato, O. Saotome and F. G. Nicodemos, "A real-time system based on FPGA to measure the transition time between tasks in a RTOS", 1st Workshop on Embedded Systems, pp. 29-39, 2010.

[16] "Low-voltage rad-hard 32-bit SPARC embedded processor TSC695FL datasheet", Atmel, 2005.

[16] "Evaluation board TSC695 user guide", www.atmel.com, 2005.

[17]http://rtemscentre.edisoft.pt/index.php?module=contentexpress&file=index&func=display&ceid=21&meid=37, acessed in 08/05/2013.

[18] “Introduction to the Quartus® II Software – Version 7.2”, Altera, 2007.

[19] http://www.hsqldb.org/

[20] http://www.r-project.org/

[21] “Stratix II Device Handbook, Volume 2 – SII5V2-4.4”, Altera, 2008.

Worst-Case Communication Overhead in a Many-Core based Shared-Memory Model

Amira Dkhil
CEA-LIST, Nano-Innov
PC172, F91191
Gif-sur-Yvette Cedex, France
amira.dkhil@cea.fr

Stéphane Louise
CEA-LIST, Nano-Innov
PC172, F91191
Gif-sur-Yvette Cedex, France
stephane.louise@cea.fr

Christine Rochange
IRIT, Université de Toulouse
118 route de Narbonne. 31062
Toulouse cedex 9, France
rochange@irit.fr

ABSTRACT

With emerging many-core architectures, using on-chip shared memories is an interesting approach because it provides high bandwidth and high throughput data exchange. Such a feature is usually implemented as a multi-bus multi-banked memory. Since predicting timing behavior is key to efficient design and verification of embedded real-time systems, the question that arises is how to evaluate the access time for one memory access of a given task while others may concurrently access the same memory-bank at the same time. In this paper, we give the answers for a subset of streaming applications modeled like CSDF Model of Computation and implemented in Kalray's MPPA chip.

Keywords

Access time, Shared memory, Multi-core, CSDF, MPPA chip

1. INTRODUCTION

Predicting timing behavior is key to efficient design and verification of embedded real-time systems. For embedded hardware platforms, Multiprocessor Systems-on-Chip (MPSoCs) provide a good balance between cost, power efficiency, and flexibility. Multi-core systems are known to be especially difficult regarding Worst-Case Execution Times (WCETs), but the new way to program these many-cores is different from the way embedded systems used to be programmed with micro-controllers: Dataflow Models of Computation (MOCs) are gaining in momentum because some subsets of the dataflow based model of computation possess good properties concerning parallelism management [4]. It has been shown recently that over 90% of streaming applications can be modeled as acyclic SDF graphs [2]. Cyclo-Static Dataflow (CSDF), the model we consider, is a generalization of SDF [8] in which consumption and production rates take the form of periodic sequences. CSDF is more versatile because it also supports algorithms with a cyclically changing, but predefined, behavior. It can provably run without locks and for well-formed applications (detectable at compilation time) in finite and statically-known memory.

Typically, these applications are partitioned into tasks that communicate over channels (i.e. FIFO buffer) together forming a Dataflow graph. In order to allow maximum flexibility at the lowest cost, tasks share storage, computation and communication resources. This leads to uncertainty about resource management which can make the system decomposable. The temporal behavior of each task becomes dependent on other tasks and cannot be analyzed in isolation. As system runs without locks, what is required to know to calculate worst-case latencies, in addition to stand-alone WCETs of individual tasks, is the communication overhead induced by the interference when several processors want to access nearly simultaneously to the same bank of shared memory. We will show that for a class of periodic scheduling schemes called implicit-deadline periodic schedule, it is possible to compute worst-case communication overhead in shared memory clusters of Kalray's MPPA chip [6] for a subset of usual stream programs with the task resulting from the compilation of the Sigma-C associated dataflow language [7]. Bamakhrama and Stefanov [2] proved that Implicit-Deadline Periodic (IDP) scheduling approach gives the maximum achievable processor utilization and throughput for large set of dataflow graphs, called matched I/O rates graphs [2]. These graphs represent more than 80% of streaming applications [2]. Self-timed schedule (STS), also known as as-soon-as possible schedule, was considered the most appropriate for streaming applications modeled as dataflow graphs [2, 3]. Moreira and Bekooij [3] established that it is possible to guarantee strictly periodic behavior of tasks within self-timed implementation. They have also provided the maximum latency for applications with periodic, sporadic and bursty sources. In [1], authors present a complete framework for computing the periodic task parameters using an estimation of worst-case execution Time. They assume that each write or read has constant execution time which is often not true.

Our approach is similar to [1, 2] in using the periodic task model which allows applying a variety of proven hard-real-time scheduling algorithms for multiprocessors. However, it is different in exploring system level in order to get upper bounds on the communication overhead as close as to the real values. Ongoing approaches focus on performance analysis either on task or system level. Especially memory accesses cannot be accurately captured on a single level alone: considering both perspectives lead to overly pessimistic estimations.

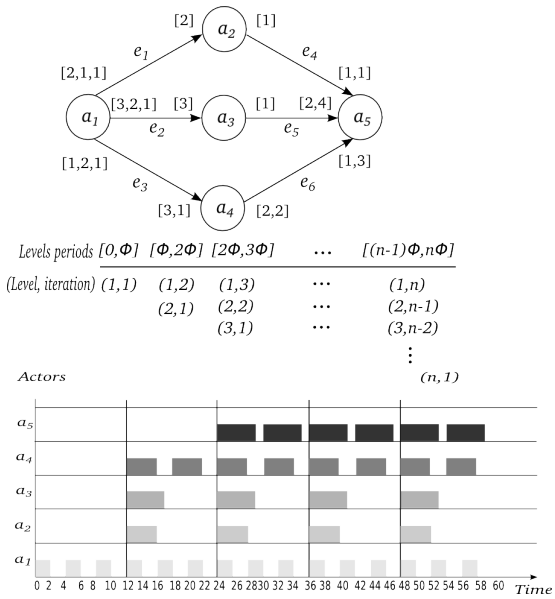


Figure 1: IDP schedule for CSDF graph

1.1 Motivating Example

We show the possible influence of parallel execution of actors, under static implicit-deadline periodic (IDP) schedule, on accesses to memory by means of an example. Figure 1 illustrates a CSDF graph consisting of five actors and six communication channels. Under IDP schedule, actors are executed in levels. Actors are assigned to levels according to the dependency and the minimum number of tokens that should be present on communication channels before firing. $A_j(n)$ is the set of actors in j level executing their n^{th} iteration. Actors in level j start execution at $t = (j-1) \times \phi$. ϕ is the global level period [1]. From the example depicted in Figure 1, at $t = 24$, actor a_5 of level 3 starts execution of his 1st iteration, actor a_1 of level 1 executes his 3rd iteration, actors a_2, a_3, $and\, a_4$ of level 2 execute their 2nd iteration. The periodic start time $t = (j-1) \times \phi$ guarantees that actors in a given level will have enough data (i.e. from their predecessors) to start. From [2], authors proved that such schedule executes with bounded memory buffers. Thus, all actors of level j can start their execution simultaneously at $t = (j-1) \times \phi$ and surely finish at $t = j \times \phi$. Well, it is simple to observe, from Figure 1, that actors can read or write memory at the same time. In this example, actors execute on five processors. A processor can request access to the shared memory without restriction but not without penalties, since there is an additional and variable arbitration cost. It would be trivial to assume that this cost is the same for all accesses to memory because this will induce a very pessimistic and maybe unreliable result. For these reasons, we must look more closely at the access modes of shared resources so that it will be possible to derive a fairly accurate estimation.

1.2 Paper Contributions

Given a streaming application modeled as an acyclic CSDF graph with periodic input streams, determine the maximum number of overlapping reads and writes when executing actors as implicit-deadline periodic tasks. For each level in the CSDF graph, define the worst-case communication overhead induced by the interference when several processors want to access simultaneously to the same bank of shared memory. The communication costs are defined as follows: 1) Arbitration cost: the time needed to arbitrate shared communication resources at run-time, 2) Synchronization cost: the time needed to check if all the necessary data is available for the actor and 3) Transfer delay: The mean-time needed to transfer input and output tokens from and to the private memory of processing elements.

The remainder of this paper is organized as follows: Section 2 introduces the CSDF model, the system model and the IDP schedule. Section 3 defines theoretical results. Finally, in section 4, we present our case study and then we conclude.

2. BACKGROUND

2.1 Cyclo-Static Data-Flow (CSDF)

We use Cyclo-Static Dataflow [9] to model real-time streaming applications. It is a directed graph $G = (A, E)$, where A is a set of computation actors and E is a set of communication channels. Data is transported in discrete chunks, called tokens, via communication channels implemented as First-In First-Out (FIFO) queues. An actor is enabled by the availability of enough tokens on each of its incoming edges. This is done by means of synhronization mechanisms like semaphores. An enabled actor can fire and consume/produce fom/to each of its input/output edges a number of tokens. Actor firings are free from side effects. Each actor $a_i \in A$ is viewed as executing through a periodically-repeating sequence of functions $[f_i(1), f_i(2), ..., f_i(\tau_i)]$ of length $\tau_i \in \mathbb{N}^\star$. P and C are the sets of production and consumption rates. For example, the jth firing of a_i is enabled if there is at least $[c_i^{e_i}(((j-1) \mod \tau_i)+1)]$ on its input channel e_i, when fired, it executes the code of function $f_i(((j-1) \mod \tau_i)+1)$ and produces $[p_i^{e_o}(((j-1) \mod \tau_i)+1)]$ tokens on its output channel e_o.

2.2 System Model

Late last year, two new architectures have emerged: the SThorm chip from STMicroelectronics (i.e. 64 cores) and MPPA chip from Kalray [6] (i.e. 256 cores). These chips rely on a clustered architecture that allows clusters of processors to share a particular level of the memory hierarchy and this has the potential to reduce the average memory access time of parallel applications [5]. A single MPPA cluster consists of 16 user processors, a controller processor, and a shared banked memory. It also comprises two DMA engines (one in and one out) to exchange data with external parts of the cluster through the NOC interface, but this is out of the scope of this paper. Each processor has private L1 cache and communicates with other processors through a shared banked memory (SRAM) of 2MB. The banked memory is implemented as a multi-bus approach [6]: it provides the same functionality as a full crossbar with lower impact on surface occupation or power consumption. Each memory-bank has a private controller which manages the requests sent from each processor in the cluster using a FIFO (first-in, first-out) equivalent queuing strategy: this will give rise to extra penalties. The shared memory is a Static RAM (SRAM), so it is quite feasible to derive some access time: some time is spent in sending a request to the controller,

and once the request is satisfied, back to the processor, this time is noted t_0. t_0 is constant because there is no memory coherence protocol. But Since we have overlapped accesses, the overall access time is:

$$t = t_0 + (\pi_j - 1) \times t_c \quad (1)$$

t_c is the time needed from the controller to access memory and it costs one RAM cycle [6]. $(\pi_j - 1)$ models the order of processor requests in the FIFO of the memory-bank controller. π_j is the number of processors executing actors of level j.

2.3 Implicit-Deadline Periodic Schedule

Under IPD static schedule, processors execute a task set $A = [A_1, A_2, ..., A_n]$ of n periodic tasks. In this paper, we consider that a task cannot be preempted during execution. A periodic task $A_j \in A$ is defined as a 4-tuple $A_j = (S_j, \omega_j, \lambda_j, D_j)$ where S_j is the start time, ω_j is the worst-case execution time, λ_j is the task period and D_j is the relative deadline of A_j. The kth invocation of task A_j is at time instants $t = S_j + k \times \lambda_j, \forall k \in \mathbb{N}$. A_j executes for ω_j time units and his execution time should not exceed D_j. A task has an implicit deadline if $D_j = \lambda_j$, it follows that A_j has to terminate before time $t = t = S_j + (k+1) \times \lambda_j$. The authors in [2] explain the following definitions in more details: Since actors of CSDF graph G are assigned to levels, we define ϕ as the minimum level period and λ as the minimum actor period. These periods are given by the solution to both equations:

$$\phi = q_1\lambda_1 = q_2\lambda_2 = ... = q_n\lambda_n \quad (2)$$

and

$$\vec{\lambda} - \vec{\omega} \geq \vec{0} \quad (3)$$

where $\vec{q} = [q_1, q_2, ..., q_n]$ is the repetition vector of G and $q_j \succ 0$ represents the number of invocations of an actor a_j in a valid schedule of G. G is consistent if there exists a repetition vector: When each actor is fired the number of times specified by $\vec{q}$, the total number of tokens produced on each arc is equal to the total number of tokens consumed.

3. WORST-CASE OVERHEAD

3.1 Assumptions and Definitions

A graph G refers to an acyclic consistent CSDF graph. A consistent graph can be executed with bounded memory buffers and no deadlock. So, we only consider consistent and deadlock free CSDFs. Consistency concerns the correspondence between production and consumption rates [11]. For our analysis, we assume the following hypothesis:

H1- Basic access time of every actor to shared memory conforms to (1). It is defined as the total time needed to access memory depending on the order of access to FIFO controller.

This order can be determined if we have exact knowledge of each access requesting time which is not feasible in practice. Under static or dynamic schedules, the order of accesses to memory cannot be determined, even for fully static schedule where we assume a very tight estimation of worst-case execution time of actors. In [12], Khandalia et al. explored the problem of imposing an ordering of interprocessor communication operations in statically scheduled multiprocessors. Their method is based on finding a linear ordering of communication actors at compile time which could minimize synchronization and arbitration costs, but this would be at the expense of some run-time flexibility. In this paper, we do not impose any constraints on communication operations.

Definition1: For a graph G under IDP schedule, the worst-case overhead O_j of level j depends on the maximum number of accesses to memory $m_{i,j}$ of actor a_i not on the exact time when a processor requests an access:
$O_j = f(m_{i,j}), \forall a_i \in A_j$. A_j is the set of actors of level j.

H2- Reading or writing tokens from/to the memory could be done at any time. This assumption was derived from simulation results: during execution, the processor may require read access when it needs some data and write access when it finishes a part of the execution.

This last assumption does not affect the considered size of shared buffers with IDP periodic schedule because reading and writing in the same shared buffer cannot be done in the same time as the execution is periodic and ordered in levels.

H4- We assume that we have reasonably tight estimates of actors computation time. Computation time is the time needed for computation operations. These estimates can be obtained by several different mechanisms like those described in [10].

H5- In [3], synchronization checks are done whenever processors communicate: the sending processor ascertains that the buffer it is writing to is not full, and the receiver ascertains that the buffer it is reading from is not empty. For IDP schedule, the synchronization cost is equal to zero, because periodic behavior guarantees that an actor a_i will finish execution before deadline D_i.

3.2 Tight overhead under IDP Model

For a simplified problem with only few processors and few concurrent tasks with few accesses to memory, the worst-case communication overhead cannot affect so much the worst-case latency of the application. The non-obvious result is that for such a configuration when the number of processors and accesses to a single-bank memory are very high, the mean access time is deeply impacted by the concurrent accesses. In this section, we introduce an execution scheme to determine an upper bound for overhead.

The CSDF graph is denoted $G_\omega = (A, E, \omega)$, ω is the set of worst-case computation time of actors. Let $S = (G_\omega, \beta, \sigma)$ be the IDP model applied to G_ω. β is the set of levels resulting from scheduling and σ is the number of levels.

Let $\beta = [\beta_1, \beta_2, ..., \beta_\sigma]$ be the set of actors for each level. $\beta_i = [\beta_i^1, \beta_i^2, ..., \beta_i^{\pi_i}]$ of level i, $\forall i \in [1, \sigma]$ is the set of actors for each level in each executing processor. π_i denotes the number of processors executing level i.

$\forall a_j \; such \; that \; a_j \in \beta_i$, it is possible to derive the associated maximum number of accesses to memory (H1). We define $M = [M_1, M_2, ..., M\sigma]$ the set of memory accesses for all levels such that $M_i = [0, m_i^1, m_i^2, ..., m_i^{\pi_i}]$ is the set of total number of memory accesses in each processor executing level i. Note that a given processor can have multiple tasks in each level.

M_i is sorted in this order: $\forall i \in [1, \sigma], \quad \forall j \in [1, \pi_i] \quad m_i^1 \leq m_i^2 \leq ... \leq m_i^{\pi_i}$, such that $m_i^{k+1} \neq m_i^k, \forall k \in [1, \pi_i]$. The new dimension of vector M_i is noted α_i. In order to get a tight overhead estimation, we assume that, for the $(\pi_i - 1)$ potential concurrent tasks on a single memory bank, the minimum number of accesses in a given level have the maximum

overhead (H2). Thus we can guarantee safe estimation. The worst-case arbitration overhead of level i is given by:
$O^i_{arb} = 0,\ if\, \pi_i = 1$
$O^i_{arb} = (m^1_i - 0) \times 1 \times t_c,\ if\, \pi_i = 2\,, \alpha_i = 3$
$O^i_{arb} = (m^1_i - 0) \times 2 \times t_c + (m^2_i - m^1_i) \times 1 \times t_c,\ if\, \pi_i = 3\,, \alpha_i = 4$
.
.
.
Where:

$$O^i_{arb} = \sum_{\delta=1}^{\alpha_i - 2} (m^{\delta+1}_i - m^{\delta}_i) \times (\pi_i - \delta) \times t_c \qquad (4)$$

Equation 4 implies that the minimum difference in the number of accesses of processors, for a given level, will get the maximum overhead and so on. This allows us to get a tight estimation of worst-case overhead since accesses will get a variable penalty. Using Equation 4, we can derive the worst-case overhead of level i by adding the transfer delay:

$$O_i = \sum_{\delta=1}^{\alpha_i - 2} (m^{\delta+1}_i - m^{\delta}_i) \times (\pi_i - \delta) \times t_c + \max_{j=1 \longmapsto \pi_i} (m^j_i \times t_0 + \omega_j) \qquad (5)$$

From (5), Equation (3) becomes:
$\vec{\lambda} - \vec{\omega} - \vec{O} \geq \vec{0}$
Thus, we can take into account communication overhead in estimating IDP periods.

4. EXPERIMENTS

We evaluated our proposed temporal analysis scheme on four programs belonging to the StreamIT Benchmarks: Direct Cosine Transform, Biotonic Sort, Audio Beam former and Laplace transformation. As architecture platform, we use the Kalary MPPA multi-cluster multi-core architecture and the associated shared memory arbitration mechanism. This paper does not consider the presence of a 2 two-way instruction and data caches for each processor of the cluster. The execution time analysis is done in three steps. First, the application is executed with a set of input data on the architecture and an execution trace of memory accesses is generated. Second, we apply the scheduling strategy in order to delimit the different phases of execution. Finally, we derive the number of memory accesses and the execution time for each node in the data-flow graph and use these informations to compute the worst-case communication overhead. The simulation time to generate these parameters was in order of minutes, the longest time was spent to derive the memory trace of each node in the application graph, because most of them contains over 60 nodes.

In the case of the MPPA cluster, π is valued between 1 and 16. As a result $(\pi_i - j)$ is bound by 15 cycles, which is the worst-case overhead associated to a given memory access. t_0 costs seven cycles [6]. For the relevant cases, in Figure 2, the worst-case overhead is between 17% and 23% of the overhead derived from simulation results.

5. CONCLUSION

The main contribution of this paper is to propose a safe timing per-cluster access memory model. This is a novel approach proposed in order to estimate the worst-case communication overhead. It was, as far as we are aware, the first time that a precise estimation of communication overhead was provided for such an architecture. The evaluations are also compliant with the experimental results. From the case study, we conclude that the timing model is very accurate and significantly improves the precision of worst-case overhead. We will also apply the same methodology for other scheduling strategies. We would like also to determine the communication overhead if accesses are distributed between the different memory banks in the same time.

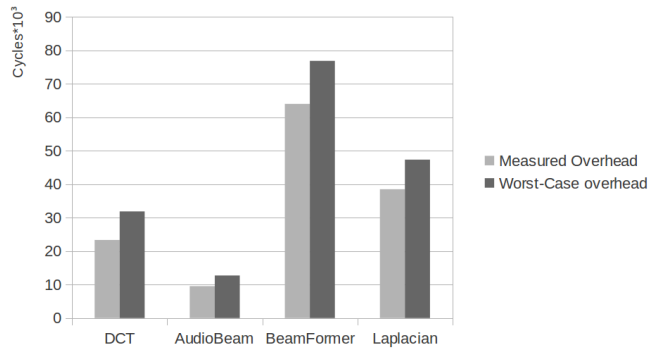


Figure 2: Measured overhead and Worst-case overhead

6. REFERENCES

[1] M. A. Bamakhrama and T. Stefanov, *Managing Latency in Embedded Streaming Applications under Hard-Real-Time Scheduling*. CODES+ISSS, 2012.

[2] M. A. Bamakhrama and T. Stefanov, *Hard-real-time scheduling of data-dependent tasks in embedded streaming applications*, EMSOFT, 2011.

[3] O.M. Moreira and M.J.G. Bekooij, *Self-Timed Scheduling Analysis for Real-Time Applications*, 2007.

[4] S.Sriram and S.S. Bhattacharyya, *Embedded Multi-processors: scheduling and synchronization*. Marcel Dekker, 2009.

[5] A. Erlichson et al. *The Benefits of Clustering in Shared Address Space Multiprocessors: An Applications-Driven Investigation*,1995.

[6] B. D. Dinechin et al., *A distributed run-time environment for the kalray mppa-256 integrated manycore processor*. ICCS Alchemy Workshop, (to be published), 2013.

[7] T. Goubier et al., *ΣC: A programming model and language for embedded manycores*, 2011.

[8] E.Lee and D. Messerschmitt, *Synchronous dataflow*. IEEE Proceedings, 1987.

[9] G. Bilsen et al., *Cyclo-static dataflow*. IEEE trans. Signal Process, Feb. 1996.

[10] R.Wilhelm et al.. The worst-case execution-time problem_overview of methods and survey of tools, 2008.

[11] E. A. Lee, *Consistency in dataflow graphs*, 1991.

[12] M. Khandelia et al., *Contention-Conscious Transaction Ordering in Multiprocessor DSP Systems*, 2006.

Towards a Programming and Analysis Framework for Timer Units

Marco Marazza
Sapienza - Università di Roma
marazza@diet.uniroma1.it

Fabio Cremona
Scuola Superiore Sant'Anna
f.cremona@sssup.it

Daniele Ceraolo Spurio
Sapienza - Università di Roma
daniele.ceraolospurio@gmail.com

Christian Nastasi
ALES S.r.l.
christian.nastasi@ales.eu.com

Carsten Demuth
STMicroelectronics Application GmbH
carsten.demuth@st.com

Alberto Ferrari
ALES S.r.l.
alberto.ferrari@ales.eu.com

ABSTRACT

Programmable Timer Units are custom co-processors, typically assembly-programmed, used to process complex high resolution timing functions subject to hard real-time constraints. The assembly language prevents code portability among Timer Units, slows code development and maintenance and limits analysis and optimization capabilities. A high-level programming and analysis framework exposing the same front-end for different Timer Units can help reducing code development time and cost. In this paper we (1) discuss our approach for achieving a high-level programming model for Timer Units, (2) present the programming model and back-end for a new Timer Unit and (3) its WCET analysis tool.

1. INTRODUCTION

More and more industrial applications demand accurate control of timing and angle synchronization. This is particularly true in automotive: common examples are power-train applications involving engine control functions where spark timing, fuel mixture control and fuel injection timing must be carefully controlled to get the highest gain in terms of fuel economy, unwanted emissions and engine performance. Software implementations based on low-latency interrupts are not always sufficient to perform high resolution timing functions subject to hard real-time constraints. Timing issues can also arise due to the large number of I/O functions to be executed in parallel. To help delivering such time-intensive functions, Timer Units can be integrated into the ECU architecture. Timer Units are customized co-processors used to offload the CPU, especially in high-end applications where single- or multi- cores are typically pushed to their limit. Examples of programmable Timer Units are the eNhanced High End Timer (NHET) [12] from Texas Instruments, the Enhanced Timer Processing Unit (ETPU) [5] from Freescale and the Generic Timer Module (GTM) [2], a new time-predictable IP developed by BOSCH. Of all the Timer Units mentioned above, only the ETPU comes with a C compiler and a Worst-Case Execution Time (WCET) analysis tool. One of the advantages of using programmable Timer Units is that these are more powerful and flexible for handling complex waveforms. However, programming in assembly has a negative impact on code optimization, time required for firmware development and code maintenance; moreover, developing code for different ECUs, each integrating a particular Timer Unit often requires implementing very similar functions via specific assembly dialects, which prevents code portability. Since Timer Units are used in safety-critical domains as e.g. automotive i.e. for engine control functions, these become integral part of so-called safety-critical systems, which are subject to the certification processes regulated by International Standards [8], [9]. Besides achieving functional requirements, safety-critical systems must meet additional non-functional requirements, i.e. related to safety, reliability and availability constraints. Evidence that non-functional requirements have been achieved can be demonstrated by means of dependability and timing analyses. Such analyses need information related to both the developed code and the specific hardware capabilities.
A framework provided with the same front-end for different Timer Units is gaining importance in the industry domain; it would allow exposing a unique set of notations to the programmer for specifying functional and non-functional behaviours, hiding the specific back-ends for object code generation and the specific tools for partial or total automation of timing analysis and dependability optimizations. This would result in considerable savings in development cost and time. In our knowledge the development of such a unified programming and analysis framework for Timer Units has not been addressed yet.

1.1 Goal and contribution of this work

The goal of this work is to achieve a high-level programming framework giving the programmer the possibility to specify functional and non-functional behaviours to program and analyse different Timer Units by means of the same set of notations. The main contributions of this work are: (1) the raising of the GTM programming model to the C abstraction level, (2) the development of a compiler back-end for the GTM IP and (3) the creation of a WCET analysis tool to analyse the GTM code. In this preliminary work we limit our study to the ETPU, one of the most common

programmable Timer Units, and the GTM, the newest one proposed by BOSCH, one of the major players in the automotive industry.
The paper is organized as follows: Sec. 2 provides a short state of the art of programmable Timer Units; Sec. 3 discusses the approach for achieving a high-level programming model for Timer Units; Sec. 4 describes how we designed the GTM programming model, built the GTM compiler back-end and the related WCET analysis tool, while Sec. 5 concludes the paper. Sec. 6 gives some highlights about our future work.

2. ARCHITECTURES

The most advanced Timer Units are provided with a custom instruction set architecture (ISA) and are usually programmed in assembly language, while simpler ones are just configurable timer arrays. The main characteristics of the ETPU [5] and the GTM [2] IPs, the most complex programmable Timer Units, are shortly described. A function performed by a programmable Timer Unit results from the combination of a hardware-implemented function (performed by configurable I/O Timer Channels) and a software-implemented one. Every I/O Timer Channel is assigned a *function* determining its behaviour. Functions are composed of so-called *threads*, which are Interrupt Service Routines (ISRs); to use a general term, from here on we will refer to these ISRs as Event Service Routines (ESRs). The GTM features up to seven processors, called Multi Channel Sequencers (MCSs). Also a MCS channel function contains ESRs. However, the execution order of ESRs is established by the ESR's code itself: MCSs are provided with instructions to synchronise the function's execution flow with: (1) time or angle events, (2) events generated by other MCS processing channels or (3) events coming from ARU-connected modules. The ARU (Advanced Routing Unit) is a module which enables data exchange among most of the GTM modules. It implements a polling-based point-to-point communication paradigm with a well-defined worst case time bound.

3. APPROACH

One of today's most commonly used programmable Timer Units is the ETPU. Hence, one of the major concerns of using the GTM IP is how to deal with the big amount of legacy code coming from applications implemented on the ETPU. The reuse of the ETPU code (or at least a part of it) on the GTM platform by means of a high level programming framework would in fact lead to considerable savings in development time and cost. Building such a framework presents the following challenges: (1) how to abstract the different hardware-implemented functionalities providing programmers with a common programming model? (2) how to move from the common representation to the different hardware/software partitionings and hardware/software interactions? (3) how to integrate into the same set of programming notations the (sub)configuration of the Channel modes? (4) how to integrate into the programming model the notations to specify non-functional requirements?
Figure 1 depicts our idea of high level programming framework for Timer Units. At the highest abstraction level the programmer should specify the model of the I/O function (independent from the Timer Unit) and the models of the hardware resources of the target Timer Units. These models

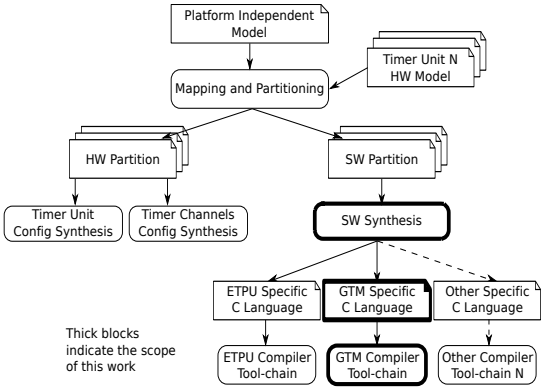


Figure 1: Common Programming Framework.

are used by partitioning and mapping algorithms to automatically generate the hardware and software partitions for the desired Timer Units. For a specific Timer Unit, the hardware partition is used to generate the configurations of its Timer Channels and the possible CPU code configuring the Timer Unit itself, while the software partition is used as input to the compiler tool-chains for synthesising the object code that will run on the Timer Unit processor(s). The software partition can be translated into C language either by an automatic tool for software synthesis or by the human programmer. The definition of platform independent models and the implementation of mapping and partitioning algorithms are complex tasks and remain out of the scope of this preliminary work. The scope of current work is highlighted in Figure 1 by thick line blocks. To limit the overall complexity while providing a high level environment for code development, we decided to build a GTM compiler yielding a programming model similar to the ETPU one. This approach has the following benefits: for those platform independent I/O function specifications resulting in the same hardware and software partitions, the mapping function to the specific Timer Unit is straightforward; if it results in different partitions, the programmer is provided with a C-like programming model to encode the software partition. Moreover, since the interfacing language is the C language for both architectures, the underlying tools remain mostly the same, except the compiler back-end, which aim is to generate the target code. Finally, having two similar programming models for different Timer Units simplifies future integration into the high level programming framework. In the following section we describe how we realized the high-level programming framework for the GTM, made of a C compiler and a WCET analysis tool for its MCSs.

4. GTM DEVELOPMENT FRAMEWORK

4.1 The GTM C-like programming model

The first contribution of this work has been the raising of the GTM programming language to the C-like level to allow for (1) human-friendly code development framework, (2) integration of WCET analysis and (3) future integration of dependability analysis and optimization techniques. Our GTM programming model (depicted in Figure 2) is based on code re-usability: it provides separation between definition and instantiation of functions. This gives the programmer the possibility to define a library of Function Prototypes

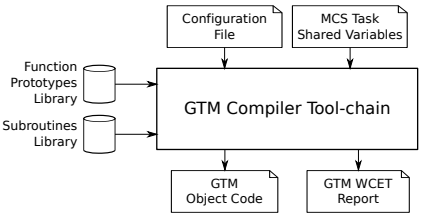


Figure 2: GTM Compiler Workflow

and then instantiate them an arbitrary number of times. A function prototype is an abstraction of collaboration of several GTM modules, including MCS processing channels, I/O Timer Channels, etc. A Function Prototype is organized according to the following structure: 1) declaration of symbolic references indicating those GTM modules required to perform the function; 2) definition of the function, including definition of its ESRs set. The Function Prototype definition is similar to a standard C function in that it accepts input parameters as an argument list. In contrast to the ETPU programming models [3], which need a specific syntax to associate each ESR with its respective activating events combination, the GTM programming model exploits ad-hoc synchronization instructions. These involve synchronization with global time and angle references, synchronization with functions running on other channels of the same MCS and synchronization with events produced by other modules connected by the Advanced Router Unit. This approach yields more readable and easier to maintain functions. Function instantiation upon different processors is achieved by defining different configuration files. A configuration file is used to produce the C-like code for a single GTM processor. Guided by the configuration file, the instantiation process consists of the following automated actions: 1) assign a name to a Function Prototype chosen from the library 2) replace the symbolic references with the appropriate addresses pertaining to the modules (e.g. input and output Timer Channels, etc.) involved in the function instance, 3) specify the processing channel for each function. To avoid name clashing when a Function Prototype is instantiated multiple times, all its variables are properly renamed.

4.2 The GTM compiler

Our GTM C Compiler has been realized upon the Clang-LLVM infrastructure [1]. The most powerful benefits of using LLVM derive from its modular nature: 1) many Timer Units' back-end can be easily added to the LLVM infrastructure and 2) compiler optimization passes operating on the compiler's Internal Representation (IR) need to be written only once for all back-ends and 3) many optimizations are already available from the Clang/LLVM infrastructure. The GTM does not support a hardware implemented stack so, to allow a function call to a subroutine, we implemented it by software; since the number of general purpose registers per Timer Channel is limited, we had also to limit the maximum number of parameters that a function call can transfer via registers. If the number of parameters exceeds the limit, these would be stored in RAM, with possible performance degradation. To obtain correct handling of instructions like *call* and *return* we had to opportunely tune the mechanisms through which LLVM translates them from the IR to the assembly code, which also involves stack management and parameter passing (calling conventions). In particular, *call* and *return* pertain to the set of instructions that are modified during the *legalization* step [1], which is the phase in which IR structures are mapped to the types and operations natively supported by the target platform. There are however some instructions that cannot be replaced during the legalization step; this is the case of IR instructions not supported by the target architecture. For this purpose, LLVM offers a different method to replace them with an equivalent set of supported instructions: the so-called *pseudo-instructions*. Pseudo-instructions are fake instructions recognized as supported by the target architecture during the legalization and instruction selection phases; these are finally replaced by a block of assembly code just before register allocation. This allows replacing a single instruction with the desired, arbitrarily complex algorithm. We used this functionality to fully support instructions such as conditional jumps, conditional moves and arithmetic shifts. Furthermore, we decided to instruct the back-end to replace the most common pseudo-instructions with subroutine calls, to avoid code repetition and save memory. However, this improvement does not always come for free, since the algorithms used to replace the instructions are software-implemented. This is the case of multiplication and division instructions: since the GTM instruction set does not support them, we implemented them by software. These studies and related architectural optimizations are left to a future study. Since the GTM instruction set architecture is very different from the general purpose ones targeted by LLVM, a few GTM processor-specific instructions (e.g. the ones for function synchronization) and some special purpose registers were not supported by the pristine IR. We decided to temporarily use the in-line assembly method to access these instructions and registers, postponing their integration to a later stage of our work. We expanded the functionality of the GTM compiler by enabling some linking capabilities: it manages allocation of functions and variables into the GTM processor memory space, avoiding waste of memory locations and address clashing.

4.3 Timing analysis tool

Timing analysis on the code designed for safety-critical hard real-time systems like Timer Units is a mandatory and crucial activity during its whole development life-cycle [8], [9]. Determining the Worst-Case Execution Time (WCET) helps bringing evidence that non functional requirements on the program execution time are not violated. Our GTM compiler integrates a tool operating static timing analyses to determine the worst-case execution time of functions executed by each GTM processor. Static methods [14] study the function's code, possibly annotated with additional information –called flow facts (*ff*)– [10], by (1) analysing the set of control-flow paths through the function, (2) combining control flow with some model of the hardware architecture and (3) obtaining upper bounds for this combination. An efficient technique performing static timing analysis is known as IPET [14]; such technique makes use of the Integer Linear Programming (ILP) approach [10]. According to the guidelines outlined in [14], the GTM can be defined as a *Time Composable Architecture*. An MCS has no caches so memory access time for data and instructions has a fixed length in terms of instruction cycles. Moreover, MCSs have

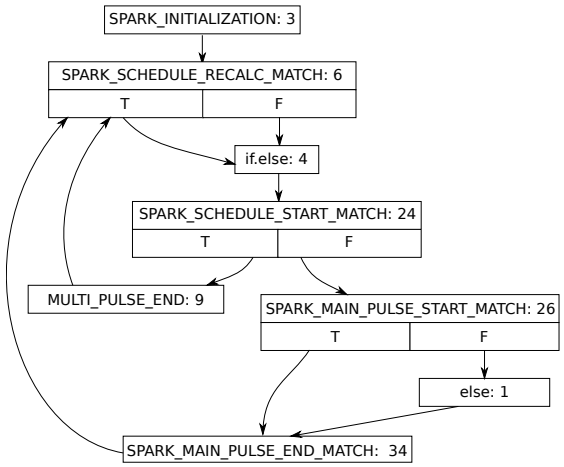


Figure 3: Back annotated CFG

a thread interleaved architecture composed by eight interleaved processing channels sharing the same pipeline, which reduces (sometimes completely removes) pipeline stalls due to pipeline hazards. This structure allows simplifying the WCET analysis: in particular, accurate microarchitecture analysis is not required. Our tool exploits a compiler analysis pass operating on the compiler IR to extract the Control Flow Graph (CFG) associated to each function. A further ad-hoc analysis pass integrated into the GTM-backend computes the length of each function's Basic Block (BB) as number of instruction cycles. CFG structure and length of BBs are combined in a back-annotated CFG (BCFG). The first analysis step is loop reduction: BCFG and *ff*s are used to reduce loop's sub-CFGs into a single BB inside the BCFG obtaining a new BCFG in the form of a Directed Acyclic Graph (DAG). Finally, the analysis proposed in [10] is applied to compute the WCET Γ:

$$\Gamma = \sum_{i=1}^{N} \gamma_{x_i} \cdot x_i \tag{1}$$

where N is the total number of BBs in the WCET path, x_i is the number of executions of a BB and γ_{x_i} is the local WCET of BB_i [10]. Figure 3 reports the BCFG obtained by implementing a simplified version of the spark plug function described in [4]. Each BB is represented as a box containing its name and local WCET information γ_{x_i} (after a colon); arrows connecting BBs indicate the possible path(s) of the control flow.

5. CONCLUSION

In this paper we pointed out that programming different Timer Units through different assembly dialects targeting the same set of applications represents the main limiting factor for code portability among Timer Units. We proposed the creation of a programming and analysis framework for Timer Units, analysed the related challenges and discussed a possible approach. As a first step we raised the programming model, developing the related C-like compiler and building the WCET analysis tool for the GTM IP, the new time predictable Timer Unit designed by BOSCH. The contribution of this work is to facilitate the migration from legacy ETPU software to the GTM one by means of a similar programming model. This encourages the next steps towards a common programming and analysis framework for Timer Units.

6. FUTURE WORK

In the future we will accomplish our GTM compiler and implement the high level programming and analysis framework for Timer Units, exploiting our GTM compiler and WCET analysis tool. Since Timer Units are often integrated in safety-critical systems, to achieve safety metrics [9, 8] we will add optimization techniques such as instruction or register duplication [6, 7], instruction scheduling [11] control flow checking [13] or register vulnerability reduction [15].

7. REFERENCES

[1] `http://llvm.org`.

[2] Gtm product information. `http://www.bosch-semiconductors.de/media/en/pdf_1/ipmodules_1/timer/bosch_product_info_gtm_ip_v1_1.pdf`.

[3] ASH WARE. *Compiler Reference Manual, version 2.01*, 12 2011.

[4] Freescale. Using the eTPU Spark Function. Application Note. `http://www.freescale.com/files/32bit/doc/app_note/AN3771.pdf`.

[5] Freescale. *ETPURM, Enhanced Time Processing Unit (eTPU) Reference Manual*, 05 2004.

[6] J. Hu, S. Wang, and S. Ziavras. In-register duplication: Exploiting narrow-width value for improving register file reliability. *Dependable Systems and Networks, DSN*, 2006.

[7] J. S. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler-directed instruction duplication for soft error detection. *Design, Automation and Test in Europe, DATE 2005*, 2005.

[8] IEC. *IEC 61508*, 04 2010.

[9] ISO. *Road vehicles – Functional safety*, 11 2011.

[10] Y. T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, 1995.

[11] S. Rehman, M. Shafique, and J. Henkel. Instruction scheduling for reliability-aware compilation. *DAC 2012, June 3-7, San Francisco, California, USA*, 2012.

[12] Texas Instruments. `http://processors.wiki.ti.com/index.php/High_End_Timer`.

[13] R. Venkatasubramanian, J. Hayes, and B. Murray. Low cost on-line fault detection using control flow assertions. *On-Line Testing Symposium, IOLTS*, 2003.

[14] Wilhelm, Reinhard et. Al. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 2008.

[15] J. Yan and W. Zhang. Compiler-guided register reliability improvement against soft errors. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 203–209. ACM, 2005.