HGT: an open-source framework for simulating parallel real-time tasks

Ignacio Sañudo, Paolo Burgio and Marko Bertogna
HiPeRT Lab, University of Modena, Italy
{ignacio.sanudoolmedo, paolo.burgio, marko.bertogna}@unimore.it

***Abstract*—With the increasing complexity of multi/many-core architectures, academy-industry research collaborations on this topic are today intensifying. In this sense, the characterization of industrial applications is still a big challenge, due to industry's reluctance to share application code details. Luckily, this trend is partly changing, and today several industrial partners can disclose high-level details of their software suites, e.g., the timing constraints or even the memory footprint and access patterns, to perform a better application characterization. However, it is still extremely cumbersome to reproduce the behavior of real applications in their actual environment, due to IPR on source code.**

In this paper, we introduce the HiPeRT Generator Tool that helps researchers creating synthetic yet realistic test cases, using a variety of techniques based on the model-driven development approach. The result is an open-source framework, that generates ready to use ANSI C code from high-level behavioral description of an application represented with a Directed acyclic graph (DAG).

I. Introduction

As of today, there is a clear trend towards the adoption of multi/many core architectures for future real-time systems to meet the stringent performance-per-energy requirements of next-generation automotive and industrial applications. Currently, researchers from academia struggle to find a way for effectively characterizing the behavior of *real* application code when running on multi- and many-cores platforms. To do so, they typically run benchmarks that stress and analyze the distinct components of the computing platforms, such as memory[1], disk[2] or CPU[3], or they use benchmarking suites to effectively validate their novel methodologies and techniques on platforms that are as similar as possible to the real ones. Significant examples from the real-time community are the Malardalen benchmark [1] and TACLeBench [2]. These benchmarks provide a collection of open-source programs to effectively validate tools and methodologies, but, unfortunately, they cannot capture the exact dynamics of *real* industrial applications.

A typical problem towards a proper validation of academic findings is that even industrial players that interested in such findings cannot share much information about the system environment, nor the source code of their applications, due to NDA and IPR restrictions. This makes it extremely difficult to replicate a realistic and representative scenario for validating research activities in the real-time system domain.

[1]http://www.bitmover.com/lmbench/
[2]https://www.coker.com.au/bonnie++/
[3]https://www.spec.org/cpu2006/

It is widely acknowledged in the real-time community that solving these issues is a key challenge for the future of research. We believe that a good solution is to encourage a "partial" collaboration between industry and academia, where the exchanged information is a high-level description of the applications behavior, that accurately captures a real scenario, without details on application internals, nor forcing companies to provide source code. These can be, for instance, the timing constraints of a set of tasks, or their memory access pattern.

To support this kind of cooperation, we implemented a tool that, starting from a behavioral description of a real-time application, automatically generates ready-to-use synthetic code that correctly mimics it. This tool, called HiPeRT Generation Tool (HGT), is implemented using a Model Driven Development (MDD) approach that, starting from text files providing a high level behavioral description of applications, generates code. HGT receives as input a set of task dependencies, timing and memory constraints represented by a Directed Acyclic Graph (DAG). Then, the constraints are parsed into an internal model and then transformed into ANSI C code, that may be executed in different target platforms.

We validate the correctness and accuracy of our tool by emulating the behavior of a real-time application specified using the UpScale DAG representation [3], produced in the P-SOCRATES FP7 project [4], and running on a x86-based system. We are currently adding support for Amalthea [5], a model adopted in the automotive domain to capture relevant task information.

The paper is organized as follows. The next section describes the Model driven development approach and the related works. Section III characterizes the task model and notation. The code generator is presented in Section IV, showing some result of the validation benchmarking in (Section V), before a concluding discussion.

II. Background

A. Model Driven Development

Model based design has been used always to create an abstract representation of the concepts. Under the Model Driven software Development (MDD) process, the model of a system under analysis undergoes multiple refinement/optimization stages to obtain a final implementation, that is behaviorally identical, but with specific properties that are amenable to designers. Such a process is typically automated through the so-called Model-to-Model Transformation (MMT) and Model to Text (M2T) transformation. In a nutshell, the code generation

process receives the model as input, and then, if necessary, it converts it in a different model with the MMT process. This can be, for instance, the conversion of a UML model into another model that better captures certain properties of the target system. Finally, the model is transformed into a lower level code representation (e.g., C/C++) through the M2T procedure. Figure 1 shows this process.

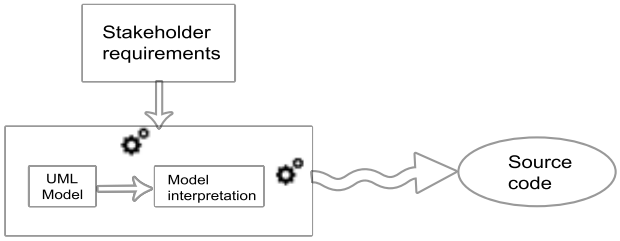

Fig. 1. Code generation process

A modeling language is composed of *syntax* and *semantic* rules [6], that define the set of language rules and the *meaning* of different language tokens. In order to create a model, MDD engineers must first define the language rules, that is, the abstract syntax or *metamodel*. A metamodel [7] is the model concept to be implemented, i.e., the textual model used for the definition of language constraints that describe a system using well-defined rules. In other words, in MDD design, the system is an instance of the metamodel.

Such an approach is extremely beneficial to software developers. First, the activities carried on in model-based software development can be easily automated, so that the software creation process becomes faster and more efficient. MDD also raises the level abstraction of the software design, allowing developers to combine lower level functionalities (e.g., drivers) as "building blocks" in a higher level of the design. Moreover, engineers are able to automatically generate code in different languages and for different platform, by only providing a solid model definition. Finally, and most importantly, this approach enables quick verification and certification of source code, enabling its adoption also in specific industrial domains.

B. Related work

There are several frameworks and tools offering the possibility to generate code from a metamodel instance. A widely used commercial tool is Simulink[4], which allows the generation of ANSI C and C++ code for real-time and non real-time applications from Matlab and Simulink diagrams. Another commercial tool is E4Coder [8]. E4Coder provides a set of tools used to simulate control algorithms and to generate code for embedded micro-controllers. The code generator tool translates ScicosLab and XCos diagrams into C language. In [9], the authors present edROOM, a graphical environment to edit ROOM models that automatically generates real-time C++ code. edROOM uses the model to describe the structure, communication topology and behavior of the system, automatically generating the application code. In [10], a MDD framework is proposed based on Java and XSLT called JComposer, for the automatic generation of real-time C-code for safety-critical embedded systems. JComposer runs on top of Linux-RTAI. In [11], a code generation framework is proposed to generate code for different target platforms modeled using AADL. In [12], another framework is presented for generating real-time code based on ADA. In this work, the functional behavior is characterized using UML2 adding the real time constraints using MARTE.

[4]https://mathworks.com/products/simulink.html

III. System model and notation

In this section, we introduce the task model and notation used throughout the paper. We consider the task model defined in [13], assuming a finite set of recurring preemptive tasks $\tau_1 ... \tau_n$. Each task is specified by a directed acyclic graph (DAG), where nodes represent task parts, and edges represent precedence constraints. Each node $\tau_{i,j}$ has an associated worst-case computation time $C_{i,j}$, corresponding to the amount of time spent for pure computation, and a worst-case memory access size $M_{i,j}$, corresponding to the dataset accessed during the node execution. Also, each task τ_i is characterized by a period T_i and a relative deadline $D_i \leq T_i$.

We also consider a hardware platform with *m* identical processors. Jobs may be scheduled using two different approaches, namely *partitioned* and *global* scheduling. Under partitioned scheduling, a task is statically allocated to a processor and task migration is not allowed, while under global scheduling, tasks can execute on any processor and migrate to a different one.

Since we plan to use our tool to test task systems complying with different memory access models, we decided to support the following execution models:

- Sparse. The "traditional" execution model, where computation and memory accesses are interleaved.
- Predictable execution model (PREM) [14], [15]. Under this model, the execution is decoupled into two different phases: a memory phase (M-Phase), during which the task pre-fetches the required data into local memory, and an execution phase (C-Phase), where the task computes the local data without accessing global memory. This technique allows improving both average and worst-case execution times, ensuring a more predictable behavior in a shared memory multi-core system, avoiding cache misses and memory interference. However, it requires full or partial code refactoring.

IV. HGT Tool

In this section, we describe the design of our HiPeRT Generator Tool. Figure 2 describes the overall system architecture and the three main components.

HGT is structured in three layers, namely the frontend, the core and the backend layer.

- The frontend layer parses input files containing the task semantics to be translated into code. These files can be of different formats.
- The core layer translates the model semantic into the HGT task model described in Section III.

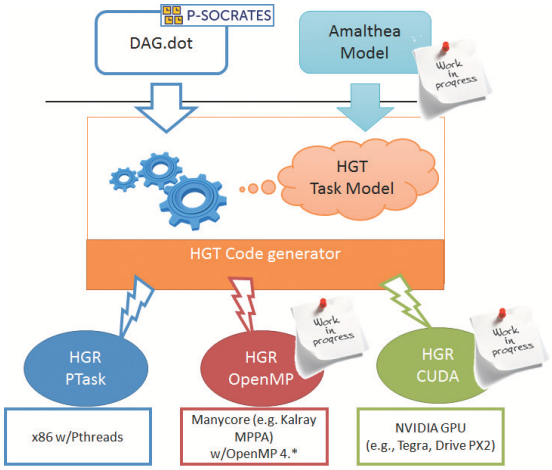

Fig. 2. HGR Hierarchy

- The backend layer generates the output (C) code. We currently support PTask/Posix threads [16], while we are developing OpenMP [17] and CUDA [18] support.

A. Front-end

The goal of the front-end layer is to read the task/system representation that expresses the task constraints, e.g, period, deadline, worst-case execution time, etc. Intuitively, the more behavioral information is provided, the more accurate is the HGT system representation. As shown in Figure 2, the front-end layer supports two languages:

- An enhanced DOT representation[5]. DOT is a graph description language used for graph representation. A DOT file is composed basically of the definition of nodes and edges, where nodes are the basic element of a graph and edges represent the precedence relationship between nodes.
- Amalthea [5], an open source format for the modeling of embedded applications, adopted by different automotive companies. Amalthea models hardware description, constraints and software requirements, with support to multi/many-core architectures.

For the sake of simplicity, we will use the semantic defined by DOT along the rest of the paper. However, both semantics are very similar as summarized in the Table I.

Unfortunately, the original specification of the DOT format does not capture real-time semantics, such as task periods or deadlines. We therefore used a specialization of the DOT format, inspired by the P-SOCRATES FP7 project [4]. We call it RT-DOT. The RT-DOT file can be either written by-hand, or generated by a compiler. In the P-SOCRATES project, this was done by the Mercurium source-to-source compiler[6].

In RT-DOT, a real time task (RT-task) is represented by a directed acyclic graph (DAG), equivalent to the model described in sectionIII. An example of a RT-DOT file is shown in Figure 3, where node *N4* will be released if and only if *N1*, *N2*, *N3* and *N5* have finished.

To be compliant with the P-SOCRATES model as well as with existing tools for the graphical representation of DOT graphs (such as GraphViz[7]), we allow the definition of multiple nodes with id '0', and we force graphical tools to ignore them (`style=invis` attribute), so they act as placeholders for task and job constraints such as period, deadline, priority. The next section explains how the task model is modified and managed by the tool.

```
digraph G2 {
      0 [period=2000, priority=60,
        deadline=2000, map=1]
      1 [WCET="10.48", MEM="256", UNIT="B"]
      2 [WCET="57.65", MEM="700", UNIT="KB"]
      3 [WCET="300.93", MEM="7", UNIT="MB"]
      4 [WCET="28.54", MEM="512", UNIT="B"]
      5 [WCET="30.78", MEM="15", UNIT="MB"]
      1 -> 2
      1 -> 3
      2 -> 4
      3 -> 4
      5 -> 4
}
```

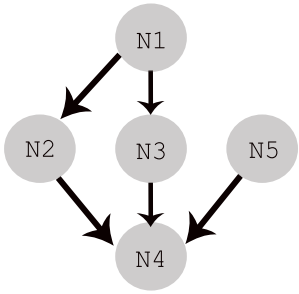

Fig. 3. Example of RT-DOT file

B. Core

The HGT core layer receives the parsed DAG representation (e.g., from RT-DOT) and does the following:

1) it determines a task grammar, in which we will precisely define the task constraints;
2) it maps the input constraints into the defined previously task grammar; and
3) it defines the rules adopted to translate the model constraints into code.

An overview of the core layer process is shown in the Figure 4.

The first step consists in creating a metamodel compliant with the constraints specified in the input file (See section II and III). Several metamodeling languages exist to support this activity, like Ecore or MOF [19]. Eclipse Modeling Framework (EMF) [20] provides a modeling framework and runtime support for the model language elaboration. EMF provides a metamodel language called *Ecore* used for the model description. It also allows the specification of attributes contained in the language and their relation through class diagrams very similar to UML diagrams.

In the second step, we map our input task model (for instance, in the RT-DOT task grammar) into our HGT Task Model as shown in Table I. In order to do so, we developed an application that parses the input model into XMI (XML Metadata Interchange). XMI is a subset of XML used for the UML model representation. We designed the HGT model to be compliant with XMI. We added parameters to the HGT task model to accurately capture the semantics of RT-tasks. These additional parameters are:

[5]https://en.wikipedia.org/wiki/DOT_(graph_description_language)
[6]https://pm.bsc.es/mcxx
[7]http://www.graphviz.org/

TABLE I
TASK MODEL EQUIVALENCES (* = WORK-IN-PROGRESS)

Front-end		**Core**	**Back-end**	
RT-DOT	**AMALTHEA**	**HGT model**	**PTask**	**OpenMP 4.5**
Graph	Task	RT-Task	Real-time PTask	Target region
Node	Runnable *	Job/Thread	PThread	Task(-part) *
Edge	–	Dependency	PThread mutex	Task dependency

1) Job execution model. It represents the memory access model, either PREM or Sparse (See section III). If in Sparse mode, it is possible to define whether the memory accesses are sequential or random, as well as the access granularity, i.e., the minimum memory-computation unit defined in bytes. Note that the use of PREM implies that memory accesses are pre-fetched "in block" into the last level cache or in a local scratchpad memory [14].
2) Task scheduling policy. Inspired by Posix Threads standard, it can be FIFO, Round Robin and CFS.
3) Partitioning policy. It may be global or partitioned. If global, jobs may migrate across cores. If partitioned, jobs are statically allocated to specific cores.

Lastly, we defined rules for parsing the HGT task model into code. With this rules, task of the HGT model are translated, e.g., into a *PTask* [16], and edges are translated into *POSIX mutex* for the ANSI C code (see Table I). We developed the front-end and core layers using the Epsilon framework [21]. Epsilon is a family of languages that provide an infrastructure for code generation, model-to-model and model-to-text transformation. It is distributed through the Eclipse platform. Specifically, the language used for translating the model into code is Epsilon Generation Language (EGL). EGL is defined by [22] as a template-based model-to-text language for generating code, documentation and other textual artifacts from models. EGL provides a template-based system in which we can define rules for the different attributes defined in the model.

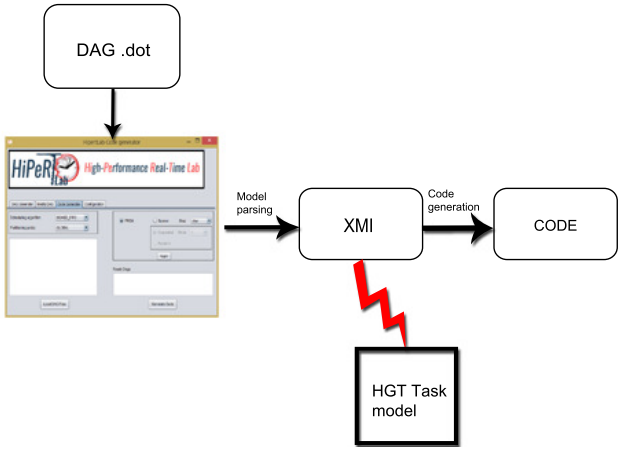

Fig. 4. Example of code generation process

C. Back-end

As we would like our HG Tool to be easily applicable to different platforms, we wanted to provide the necessary software abstraction to transparently support execution of **the same code** across different architectures.This can be difficult because, e.g., not all platforms support the PThread [23] or the OpenMP API [17]. A corner case is represented by heterogeneous platforms (e.g., based on discrete GPUs, or on many-core accelerators), that do not provide a shared memory abstraction [24], [25].

For this reason, our tool generate code that runs on top of a layer, called *HG-Runtime* (HGR), that can be ported on different platforms. HGR exposes a very simple APIs that allows:

- generating RT-tasks with their own period, deadline, priority, etc;
- spawning RT-threads/jobs within a single task, and managing their dependencies;
- simulating the execution of 'C' clock cycles on a platform core, with or without performing memory accesses (simulating cache misses);
- accessing 'M' bytes in the main platform memory.

For space reasons, we do not provide all details of the HGR API in this paper.

We currently implemented a first version of HGR that is based on the PTask library [16], a research API that enhances PThreads with real-time characteristics, like task periods, deadlines, priorities, and OS scheduling policy. Our runtime has been validated on a laptop running Linux with real-time extensions (see Section V). We also plan to target more realistic many-core accelerators such as the Kalray MPPA[26], [4], by implementing HGR on top of the OpenMP runtime developed specifically for that platform during the P-SOCRATES project. We will also support automotive-grade heterogeneous SoCs based on tightly-coupled NVIDIA GPUs [24], [25], porting the HGR API on top of CUDA [18].

V. IMPLEMENTATION

In this section, we show how we implemented the synthetic tasks generated by the HGT tool. The main problem was to accurately model memory and execution phases. We ran different experiments, measuring the error between the expected (M — C) emulated time and the actual one. Measures are taken on an i7-4770T CPU @ 2.50GHz, with 32GB of RAM, running on an standard Ubuntu 4.4.0-53.

A. Memory Phase

We first compared the memory transfer delay of different *memcopy* implementations, namely:

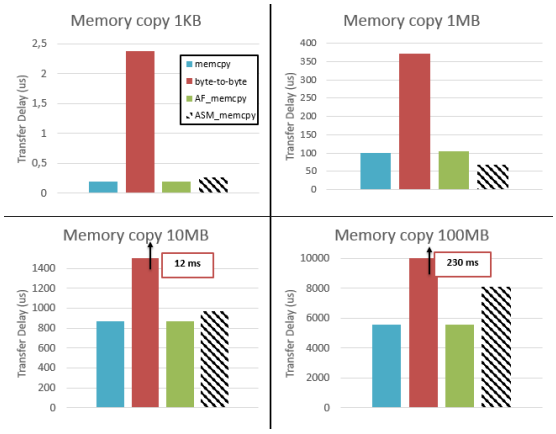

Fig. 5. Memory test

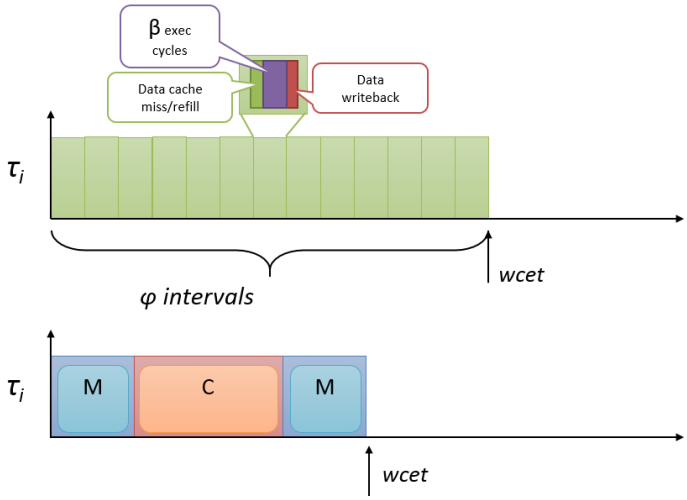

Fig. 6. Task execution model

- the standard version of `memcpy`;
- a byte-to-byte copy;
- *AFmemcpy*[8], an optimization of the `memcpy` written in assembly; and
- ASM memcpy, our implementation of the memcpy written in assembly.

As shown in Figure 5, the standard implementation of the AF memcpy performs slightly better than the "standard" `memcpy`. Still, to avoid compatibility issues, we used the former implementation. For the very same reason, it is not worth implementing (non-portable) ASM code for the M phase, even if in some cases it might be the best performing one.

B. Execution Phase

As explained in section IV, the front-end provides a DAG where each node is characterized by a worst-case execution time, specified in time-units, and a memory access size, specified in bytes. Depending on the execution model adopted, PREM or sparse, the HGT implementation of memory accesses at node level varies. Under the PREM model, memory phases are implemented using a single `memcpy` of corresponding size, followed by an execution phase lasting for the specified WCET. Under the sparse model, instead, we decided to evenly divide the memory accesses into multiple sequential blocks, each accessing memory with a given *granularity* (specified in the front-end). The number of blocks for a node $\tau_{i,j}$ can therefore be computed as

$$\phi_{i,j} = M_{i,j}/granularity. \quad (1)$$

Similarly, the worst-case execution time of the considered node is accordingly distributed among the blocks, so that each block has an execution time of $\beta_{i,j}$, where

$$\beta_{i,j} = C_{i,j}/\phi_{i,j}. \quad (2)$$

An example of both approaches is illustrated in the Figure 6.

We experimented two different ways to reproduce a given execution time: ASM and CLOCK. The ASM implementation executes $\beta_{i,j} * F$ NOP ALU operations, coded in assembly, where F is the CPU frequency. The CLOCK implementation employs a spinning approach where the task continuously reads a timer to check when $\beta_{i,j}$ time units have elapsed. To compare the effectiveness of both approaches, we characterized the difference between the expected and the measured execution times under different configurations, i.e., for different $C_{i,j}$-$\phi_{i,j}$ combinations. Each configuration was ran 10K times, measuring the largest divergence w.r.t. the desired value. Results are summarized in Figure 7, showing the percentage of the accuracy error.

	PHI	512	1024	10240	102400	1024k	10240k	102400k
1 ms	CLOCK	6,10	10,27	89,37	889,25	9598,57	96614,45	966825,65
	ASM	2,70	3,97	11,39	32,60	78,19	1667,25	17589,36
10 ms	CLOCK	0,90	1,82	10,75	86,05	879,84	9572,47	96624,53
	ASM	3,71	2,15	2,47	9,47	13,73	76,47	1664,93
100 ms	CLOCK	0,11	0,23	1,22	8,61	83,31	866,89	9569,64
	ASM	0,62	0,48	0,79	1,07	8,19	11,60	73,29
500 ms	CLOCK	0,05	0,09	0,33	2,07	16,72	141,41	1833,61
	ASM	0,39	0,33	0,34	0,48	1,87	15,41	17,22
1 sec	CLOCK	0,03	0,05	0,19	1,06	8,41	82,54	866,99
	ASM	0,40	0,39	0,22	0,34	0,95	8,05	10,97
2 sec	CLOCK	0,00	0,01	0,07	0,58	4,31	46,79	383,48
	ASM	0,41	0,20	0,23	0,28	0,54	4,21	38,18

Fig. 7. Computation test

The most accurate approach is the ASM. Still, there are problems when reproducing tasks with a small execution time and a large data size (thus, a small C/ϕ ratio). A reasonable accuracy can be reached for block sizes above 1us. We are working on more enhanced ASM methods that may obtain a higher accuracy also for smaller block sizes. The drawback of these alternative approaches is that they are not platform-independent, but they require a re-engineering when changing architecture.

VI. Conclusion

We presented an open-source tool for generating synthetic real-time tasks complying with different memory and execution models: parallel or sequential, DAG-based or Autosar,

[8]url: http://www.agner.org/optimize

PREM or sparse. The tool has been designed following a model-driven development approach, to allow for an easier extensibility and customization to different hardware and software architectures. The generated code can be adopted to test the effectiveness of scheduling algorithms, operating systems and runtimes under a variety of configurable workloads, allowing one to test the impact of different execution models over a considered architecture.

In the future, we plan to use this tool to generate synthetic benchmarks to test the effectiveness of PREM-based execution models with respect to standard approaches under different multi/many-core architectures and operating systems. The front-end will be extended to be compatible with the Amalthea model, while the back-end will be ported on top of the OpenMP/CUDA runtimes supported by Kalray MPPA and NVIDIA GPUs. Finally, the tool will be enhanced to include a configurable number of shared resources and critical sections for more realistically modeling real industrial applications.

The source code and the tool may be downloaded from our website[9].

ACKNOWLEDGMENT

This work was supported by the HERCULES Project, funded by European Union's Horizon 2020 research and innovation program under grant agreement No. 688860

REFERENCES

[1] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET Benchmarks: Past, Present And Future," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, ser. OpenAccess Series in Informatics (OASIcs), B. Lisper, Ed., vol. 15. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 136–146, the printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2010/2833

[2] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, and C. R. et al, "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, ser. OpenAccess Series in Informatics (OASIcs), M. Schoeberl, Ed., vol. 55. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 1–10. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6895

[3] The P-SOCRATES Consortium, "UpScale SDK – Unleashing the power of high-performance in real-time applications." [Online]. Available: http://www.upscale-sdk.com/

[4] L. M. Pinho, V. Nélis, P. M. Yomsi, E. Quiñones, M. Bertogna, P. Burgio, A. Marongiu, C. Scordino, P. Gai, M. Ramponi, and M. Mardiak, "P-SOCRATES: A parallel software framework for time-critical many-core systems," *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 39, no. 8, pp. 1190–1203, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.micpro.2015.06.004

[5] Amalthea, "Amalthea. model based open source development environment for automotive multi core systems," 2014. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6895

[6] G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, Eds., *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4735. Springer, 2007.

[7] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*, 1st ed. Addison-Wesley Professional, 2008.

[8] E4Coder, "The toolset for simulation and code generation for embedded devices." [Online]. Available: http://www.e4coder.com/

[9] P. Parra, A. Viana, O. Rodríguez, M. Knoblauch, F. Alcojor, S. Sánchez, I. García, O. García, and D. Meziat, "Edroom. automatic c++ code generator for real time systems modelled with room," in *Actas del Taller sobre Desarrollo de Software Dirigido por Modelos. MDA y Aplicaciones. Sitges, Spain, October 3, 2006*, 2006. [Online]. Available: http://ceur-ws.org/Vol-227/paper12.pdf

[10] L. Carnevali, D. D'Amico, L. Ridi, and E. Vicario, "Automatic code generation from real-time systems specifications," in *Proceedings of the Twentienth IEEE/IFIP International Symposium on Rapid System Prototyping, Shortening the Path from Specification to Prototype, RSP 2009, Paris, France, 23-26 June 2009*, 2009, pp. 102–105. [Online]. Available: http://dx.doi.org/10.1109/RSP.2009.24

[11] B. Kim, L. T. X. Phan, O. Sokolsky, and I. Lee, "Platform-dependent code generation for embedded real-time software," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES 2013, Montreal, QC, Canada, September 29 - October 4, 2013*, 2013, pp. 8:1–8:10. [Online]. Available: http://dx.doi.org/10.1109/CASES.2013.6662512

[12] E. Salazar, A. Alonso, M. A. de Miguel, and J. A. de la Puente, "A model-based framework for developing real-time safety ada systems," in *Reliable Software Technologies - Ada-Europe 2013, 18th Ada-Europe International Conference on Reliable Software Technologies, Berlin, Germany, June 10-14, 2013. Proceedings*, 2013, pp. 127–142. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38601-5_9

[13] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Schedulability analysis of conditional parallel task graphs in multicore systems," *IEEE Trans. Computers*, vol. 66, no. 2, pp. 339–353, 2017. [Online]. Available: http://dx.doi.org/10.1109/TC.2016.2584064

[14] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*, 2011, pp. 269–279. [Online]. Available: http://dx.doi.org/10.1109/RTAS.2011.33

[15] P. Burgio, A. Marongiu, P. Valente, and M. Bertogna, "A memory-centric approach to enable timing-predictability within embedded many-core accelerators," in *2015 CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*, Oct 2015, pp. 1–8.

[16] G. C. Buttazzo and G. Lipari, "Ptask: An educational C library for programming real-time systems on linux," in *Proceedings of 2013 IEEE 18th Conference on Emerging Technologies & Factory Automation, ETFA 2013, Cagliari, Italy, September 10-13, 2013*, 2013, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/ETFA.2013.6648001

[17] "OpenMP Application Program Interface v4," 2011. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[18] "CUDA Toolkit Documentation v7.0," http://docs.nvidia.com/cuda/index.html, accessed: July, 30th 2015.

[19] OMG, "Mof 2.0/xmi mapping v. 2.1.1. technical report, object management group," 2007.

[20] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[21] A. G.-D. R. P. Dimitris Kolovos, Louis Rose, *The Epsilon Book. The Eclipse Foundation.*, 2014.

[22] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. C. Polack, *The Epsilon Generation Language*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–16. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69100-6_1

[23] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[24] NVIDIA, "Nvidia tegra x1 white paper, nvidiaś new mobile superchip," *NVIDIA Corporation*, 2015. [Online]. Available: http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf

[25] ——, "Nvidia tegra k1 white paper, a new era in mobile computing," *NVIDIA Corporation*, 2014. [Online]. Available: http://www.nvidia.com/content/pdf/tegra_white_papers/tegra_k1_whitepaper_v1.0.pdf

[26] Kalray Corporation, "Many-core Kalray MPPA," [Online] http://www.kalray.eu/, 2012.

[9] https://github.com/nachoSO/hipert.hg