HIPEAC 2019 Conference

Proceedings of

RAPDIO 2019 Workshop

Valencia, Spain

22th January 2019

#HiPEAC19

January 21-23, 2019, Valencia, Spain

Organizing committee

Daniel Chillet, *University of Rennes 1*
Reda Nouacer, *CEA List*
Morteza Biglari-Abhari, *University of Auckland*
Daniel Gracia Pérez, *Thales Research & Technology*
Gianluca Palermo, *Politecnico di Milano*

Program committee

Mario Porrmann, *Bielefeld University*
Roberto Giorgi, *University of Siena*
Philipp A. Hartmann, *Intel*
Jeronimo Castrillon, *TU Dresden*
Sotirios Xydis, *National Technical University of Athens*
Michael Huebner, *Ruhr-University Bochum*
Tim Kogel, *Synopsys*
Frédéric Pétrot, *TIMA Lab, Grenoble Institute of Technology*
Antonino Tumeo, *Politecnico di Milano*
Pierre Boulet, *Univ Lille 1, CRIStAL*
Davide Quaglia, *University of Verona*
Christian Haubelt, *University of Rostock*
Alper Sen, *Bogazici University*

Website

`http://www.rapido.deib.polimi.it`
`http://www.rapido.deib.polimi.it/RapidoProceedings.pdf`

RAPIDO

Schedule

— **Session 1** 10 : 00 − 10 : 55
- — **Keynote 1 Andreas Herkersdorf**, Technische Universität München
  Self-aware/self-organizing MPSoC on the basis of predictable machine learning

— **Session 2a** 11 : 30 − 12 : 15
- — **Keynote 2 Anton Lokhmotov**, Dividiti
  Community-driven benchmarking of AI/ML systems ; the ReQuEST and MLPerf initiatives

— **Session 2b** 12 : 15 − 12 : 55
- — Bewoayia Kebianyor, Philipp Ittershagen and Kim Gruettner
  Towards Stateflow Model Aware Debugging with LLDB
- — Lukas Jünger, Jan Henrik Weinstock, Rainer Leupers and Gerd Ascheid
  Fast SystemC Processor Models with Unicorn

— **Session 3a** 14 : 00 − 14 : 45
- — **Keynote 3 Amir Charif**, CEA LIST
  Ultra-fast virtualization and simulation tools to meet the future needs of the industry

— **Session 3b** 14 : 45 − 15 : 25
- — Amir Charif, Gabriel Busnot, Rania Mameesh, Tanguy Sassolas and Nicolas Ventroux
  Fast Virtual Prototyping for Embedded Computing Systems Design and Exploration
- — Rodrigo Cortés Porto, Daniela Genius and Ludovic Apvrille
  Modeling and Virtual Prototyping for Embedded Systems on Mixed-Signal Multicores

— **Session 4a** 16 : 00 − 16 : 45
- — **Keynote 4 Katerina Slaninova**, IT4Innovation
  Advanced Simulator for Smart City Traffic Flow Optimisation

— **Session 4b** 16 : 45 − 17 : 25
- — Abdallah Saad, Ahmed El-Mahdy and Hisham El-Shishiny
  Performance Modeling of MPI-based Applications on Cloud Multicore Servers
- — Roberto Giorgi, Marco Procaccini and Farnam Maybodi Khalili
  A Design Space Exploration Tool Set for Future Tera-scale High-Performance Computers

List of regular papers

Keynotes

Keynote 1 **10 :00 - 10 :55**

Self-Adaptive MPSoC Runtime Optimization

— *Andreas Herkersdorf, Technische Universität München*

— **Abstract :** Multicore technology plays a pivotal role for conquering key societal challenges. Safe, ecological mobility, wide-spread rollout of e-health, smart industrial automation and the development of a secure, high-bandwidth, low-latency mobile communication infrastructure, all these cyber physical application domains critically depend on high-performance, low power and dependable computing. The hardware / software complexity of these advanced computing platforms, with Millions of lines of code, tens of processing cores, heterogeneous accelerators, diverse I/O and memory interfaces, span a design space which no longer allows for exhaustive exploration with classical design time simulation and formal verification methods. However, that's not yet it. Platform uncertainties, arising from variability in semiconductor production, long term aging effects and/or imprecise models, as well as application uncertainties, either from a lack of application knowledge or short term application load dynamics, further increase uncertainties in the expected / required system behavior.
Instead of using a limited number of pre-defined system operating points determined at design time, we suggest employing run-time architecture adaptation by controlling platform dynamics with a combination of self-aware/self-organizing machine learning techniques with formal reactive methods in order to provide platform worst-case real-time and safety guarantees. The MPSoC shall be enabled to autonomously adjust critical operation parameters, such as core frequency, supply voltage, task to core mapping, for the sake of working around permanent and transient defects and adjusting to varying environmental conditions and workloads. We envision two layers of hierarchical control, a hardware-based reinforcement machine learning entity consisting of learning classifier tables and a software-based supervisory control for handling system-wide objectives on longer time intervals. A proof of concept for the approach has been achieved with SystemC-based simulations and FPGA prototype implementation measurements for video processing and IP (Internet Packet) forwarding applications. The presented concept represents ongoing collaborative work by UC Irvine, US, TU Braunschweig and TU Munich, DE, in which we combine our past projects experience with self-awareness and self-organization at hardware and software layers of cyber physical multicore platforms.

— **Biography :** Andreas Herkersdorf is a professor in the Department of Electrical and Computer Engineering and also affiliated to the Department of Informatics at Technical University of Munich (TUM). He received a Dr. degree from ETH Zurich, Switzerland, in Electrical Engineering in 1991. Between 1988 and 2003, he has been in technical and management positions with the IBM Research Laboratory in Rüschlikon, Switzerland. Since 2003, Prof. Herkersdorf leads the Chair of Integrated Systems at TUM. Between 2014 and 2017, he served as Dean of Study Affairs in the Department of Electrical and Computer Engineering. He is a senior member of the IEEE, elected member of the DFG (German Research Foundation) Review Board, advisor of the Bavarian government for the initiative "Smart Innovations" and serves as editor for Springer and De Gruyter journals for design automation and information technology. His research interests include application-specific multi-processor architectures, IP network processing, Network on Chip and self-adaptive fault-tolerant computing.

Keynote 2 **11 :30 - 12 :15**

Community-driven benchmarking of AI/ML systems ; the ReQuEST and MLPerf initiatives

— *Anton Lokhmotov, Dividiti*

— **Abstract :** We've been crusading for community-driven benchmarking for over a decade, so we are pleased to see that kindred ideas are beginning to capture imagination of a broad community interested in benchmarking AI/ML systems (models/software/hardware). As we see it, community involvement implies using at least three good things : representative workloads, rigorous and fair methodology, and state-of-the-art workflow automation. In this talk, I'll describe our experience with organizing the first Reproducible Quality-Efficient Systems Tournament (ReQuEST @ ASPLOS'18 : http ://cknowledge.org/request), and contributing to the new MLPerf initiative (http ://mlperf.org).

— **Biography :** Dr Lokhmotov has been working on optimising computer systems for over 15 years, both as a researcher and engineer. He is CEO of dividiti. In 2015, Dr Lokhmotov co-founded dividiti to pursue a vision of efficient and reliable computing everywhere.
In 2010-2015, Dr Lokhmotov led development of GPU Compute programming technologies for the ARM Mali GPU series, including production (OpenCL, RenderScript) and research (EU-funded project CARP) compilers. He was actively involved in educating academic and professional developers, engaging with partners and customers, contributing to open-source projects and standardisation efforts. In 2008-2009, as a research associate at Imperial College London, Dr Lokhmotov investigated productivity, efficiency and portability of programming techniques for heterogeneous systems.
Dr Lokhmotov received a PhD in Computer Science from the University of Cambridge in 2008, and an MSc in Applied Mathematics and Physics (summa cum laude) from the Moscow Institute for Physics and Technology in 2004.

Keynote 3 — 14 :00 - 14 :45

Ultra-fast virtualization and simulation solutions to meet the current and future needs of the industry

— *Amir Charif, CEA LIST*

— **Abstract :** To enable rapid early prototyping of modern computing and cyber-physical systems, which are continuously growing in both complexity and scale, two factors come into play : The time required to build a working prototype of the system, and the time required to evaluate it. This talk introduces a complete prototyping and architectural exploration framework to respond to these needs. It speeds up the construction of complete early prototypes by maximizing model reuse and enabling interoperability between various prototyping environments and modelling abstractions (e.g. QEMU, SystemC/TLM, RTL, FPGA emulators, FMI-compatible physical simulators, etc.). The entire system is described and configured using a single high-level description, which abstracts away the underlying tools, models and communication protocols to reduce the platform setup effort. Second, several methods for accelerating simulations, including kernel-level parallelism for SystemC subsystems and type decoupling between subsystems are explored. Perspectives on future improvements and research directions will also be presented in this talk.

— **Biography :** Amir Charif is a research-engineer at CEA LIST. He obtained his Engineer's degree in Electronics and Embedded Computing from Polytech'Grenoble in 2014, and his PhD in Nanoelectronics and Nanotechnology from the National Polytechnical Institute (INP) of Grenoble in 2017. He has published many contributions to the fields of deadlock-free routing in 2D and 3D MPSoCs, parallel cycle-accurate simulation of NoC-based systems and NoC reliability. In 2017, he joined the CEA LIST to work on the fast simulation and prototyping of complex cyber-physical systems.

Keynote 4 — 16 :00 - 16 :45

Advanced Simulator for Smart City Traffic Flow Optimisation

— *Katerina Slaninova, IT4Innovation*

— **Abstract :** Advanced Simulator for Smart City Traffic Flow Optimisation Abstract : Optimisation of traffic flow is one of the topics discussed within the Smart City domain. We will present an enhanced

real-time traffic simulator running on High Performance Computing infrastructure for testing efficiency and usability of a self-adaptive navigation system which implements a traffic flow optimization service. Building blocks of the simulator include a server-side navigation system, Virtual Smart City World, benchmark settings, and simplified simulation of vehicles. The important feature of the simulator is the ability to evaluate the traffic flow control strategy in the Smart City world, both with and without enabled Global View calculation of a traffic network for a given percentage of vehicles connected to the server-side service.

— **Biography :** Dr. Katerina Slaninova is currently Deputy head of Advanced Data Analysis and Simulation Lab at IT4Innovations and Assistant Professor at Silesian University of Opava, Czech Republic. Her research interests include informational retrieval, traffic analysis, data mining, process mining, and complex networks. She is co-investigator of H2020-FET HPC projects ANTAREX (AutoTuning and Adaptivity appRoach for Energy efficient eXascale HPC systems) where she worked within the team of the Center for the Development of Transportation Systems ?- RODOS - and participates in long-term cooperation with the navigation company Sygic, Slovakia. She has published more than 50 articles in scientific international journals, book chapters, and conferences in her research field.

Towards Stateflow Model Aware Debugging with LLDB

Bewoayia Kebianyor, Philipp Ittershagen and Kim Gruettner

Towards Stateflow Model Aware Debugging with LLDB

Bewoayia Kebianyor
Philipp Ittershagen
Kim Grüttner
bewoayia.kebianyor@offis.de
philipp.ittershagen@offis.de
kim.gruettner@offis.de
OFFIS – Institute for Information Technology
Oldenburg, Germany

ABSTRACT

In many of today's products the embedded software is designed and tested in a model based environment. With the help of code generation techniques, a part of the model can be cross-compiled and integrated on the processor of the final product. While testing and debugging of the input model is usually well supported, debugging of the generated code for the target processor together with its run-time environment is more difficult, because traceability between the generated code and the input model is lost or has to be manually reconstructed. This paper focuses on providing traceability of automatically generated source code to model elements. Our proposed approach extends LLDB with model related debug information to become aware of the original input model. We demonstrate the correct functionality of our approach by running the model-related LLDB commands for two Stateflow models with generated source code in C and C++ respectively. The concept presented here is not limited to Stateflow models, but can be applicable to other models where source code is automatically generated with source-to-model traceability tags.

KEYWORDS

Traceability, Model-level Debugging, LLDB

ACM Reference Format:
Bewoayia Kebianyor, Philipp Ittershagen, and Kim Grüttner. 2019. Towards Stateflow Model Aware Debugging with LLDB. In *Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '19), January 21–23, 2019, Valencia, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3300189.3300190

RAPIDO '19, January 21–23, 2019, Valencia, Spain

ACM ISBN 978-1-4503-6260-3/19/01…$15.00
https://doi.org/10.1145/3300189.3300190

1 INTRODUCTION AND MOTIVATION

The development of today's technical systems and applications are becoming more and more complex as the demand for compact devices with shrinking size of processors and microchips increase. Combined with the increasing need to develop high-quality products with a short time-to-market goal, this leads many companies to employ model-based software development to cope with these challenges. The increase in complexity of these devices means an increase in the complexity of the software running on them, and an increase in unwanted behavior. Model-based software development provides the benefits of abstraction and automation. It can be combined with automatic generation of software source code which reduces the introduction of faults into the system. Nevertheless an error-free software cannot be guaranteed. Therefore, the need for error detection at all levels of the software development process becomes a very important part. Since the generated source code strictly depends on the input model, providing traceability of software defects back to the input model-level is invaluable.

Several approaches have been proposed towards model-level debugging (see Sec. 2). Some approaches are based on debugging in a simulated environment, while in some [2, 5] when the source code is generated, the code is instrumented with trace functions. During execution at run-time, trace data are captured and mapped to the model when debugging. Code instrumentation generally increases the size of the executable and thereforre impacts the run-time behavior of the system. Such approaches additionally often times require proprietary tracing infrastructure on the target platform in contrast to a gdb-server-like de-facto standard debugging interface.

Some commercial tools like Matlab/Simulink or Enterprise Architect/Embedded Engineer that support model-level debugging offer rapid-prototyping through model-based design stages which can be identified as model-in-the-loop (MIL), software-in-the-loop (SIL) and virtual-platform-in-the-loop (VPIL) simulation models. These steps offer very good debugging support and an extensive control over the environment on the host machine. However, the final step of the model-driven developement features a hardware-in-the-loop (HIL) setup, where the model is cross-compiled and executed on the target platform combined with a (possibly refined) environment model. Due to the cross compilation stage and the reduced traceability of the model execution on the platform,

the link between the high-level model description and the executed binary is lost.

In this paper, we propose a platform-independent model-based debugging approach for HIL setups that makes use of existing open-source software technologies and is neither simulation-based nor instrumentation-based. Our approach provides the benefit of observing the model behavior in the generated and cross-compiled code, either on an instruction set simulator or on the real target processor. This approach enables native debugging of the model after its integration into the target processor environment, thereby especially considering its refined interface behaviour and the interaction with the external environment.

Existing open source tools like LLDB, Clang and Xerces (C++ XML Parser) are leveraged to achieve this goal. We investigated our approach by debugging a Stateflow model using LLDB. However the approach can be applied to different model types for which production C/C++ source code is generated with embedded code-model traceability tags.

This paper is organized as follows: Sec. 2 discusses related work. Sec. 3 describes used background technologies, Sec. 4 presents the overall concept of our approach, and in Sec. 5 we present the implementation of the concept described in Sec. 4 for an example Stateflow model. In Sec. 6 we present the experimental results and finally Sec. 7 concludes this work and provides a short outlook on future work.

2 RELATED WORK

Several approaches towards model-level debugging have been proposed and some few commercial CASE tools exist for debugging at the model level. These can be categorized into host and target debugging. Host debugging refers to model-level debugging in a simulated environment while target debugging describes debugging on a target system. Target debugging can further be categorized into debugging with gdb facilities and debugging with a proprietary trace communication.

2.1 Host Based Debugging

Some approaches are based on model-checking [1] and simulation [2, 4, 7, 8]. Matlab/Simulink is one of the commonly used commercial tools for model driven software development in the area of embedded system design. It provides debugging at the model-level via simulation, as well as debugging the generated code via Software-in-the-Loop simulation [7]. Matlab/Simulink also supports real time testing (Rapid Prototyping and Hardware-in-the-Loop) [8]. Another approach discussed in [2] is the Rational Rhapsody. Rhapsody generates code with instrumentation from the model and uses its animation feature to validate the model by tracing and simulating the executable model.

2.2 Target Based Debugging

In [4] an approach based on generating and instrumenting code for different abstraction levels of an embedded system is described. At run-time trace data are collected and mapped back to the model. Debugging is then performed at model-level by visualizing the input data. Although this approach provides a mapping of run-time data from the platform to the model level, the code instrumentation increases the size of the code and affects the execution time in a production code. This approach (as in the Rhapsody approach) is in-efficient for small target systems due to limited resources.

The approach in [3] focuses on generating code with model-to-code traceability tags for State-chart models. During debugging, program slicing techniques are used on the generated code to identify a reduced program (Slice) responsible for an unwanted behavior. The program slice is then related to State-chart using the traceability tags. However, this approach is limited as program slicing will not work where there are faults due to errors of omission, for example missing variable initialization etc.

In the presented approaches, debugging of an application on target system with both source-level and model-level debugging is not fully supported. This paper addresses the combination of source-level and model-level debugging of an application running on a target platform using a de-facto gdb-like debugger.

3 BACKGROUND

3.1 Clang

Clang is a C language family (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript) front-end for the LLVM compiler and tool chain project. It has a modular design and library based architecture which separates different parts of the front-end into different libraries with well defined interfaces (API)[10].

Like most compilers, Clang is able to parse and analyze any source code written in the C language family into an Abstract Syntax Tree (AST) for representing the source code. Applications can interface to Clang using the LibClang, Clang Plugin or LibTooling interface depending on their specific needs.

LibClang is a stable high level C interface to Clang and with backward compatibility. It uses a cursor for interacting the AST, but without support for full control of the AST.

Clang Plugins are dynamic libraries loaded at run-time and provide possibility to run additional actions on the AST during compilation. Supports full control over the Clang AST.

LibTooling provides a C++ interface to be used in writing standalone tools and full control over the Clang AST

3.2 LLDB

LLDB is a gdb-like debugger software of the LLVM project. It is a high performance debugger, built as a set of reusable components which highly leverage existing libraries within the LLVM project [11]. Its architecture supports debugging modern multi-threaded programs , and the effective handling of debug symbols. It has a plug-in support for functionality, portability and extensibility. LLDB provides a public interface

to applications: the SB APIs for C++ applications and script bridging interface for usage within the LLDB embedded script interpreter, and also in any python script.

The support for user-defined plugins is the key feature utilized to extend LLDB for model-aware debugging. Using the LLDB API, we defined model-related commands and added them to the command handler. With the plugin infrastructure we provided a plugin as a command-line shared object which is loaded at run-time to extend the LLDB with model-related debug commands. Via the model-related debug commands, LLDB gains access to the model-level debugging information. Implementation details are explained in Sec. 5.

3.3 Stateflow, Matlab Simulink Coder and Embebded Coder

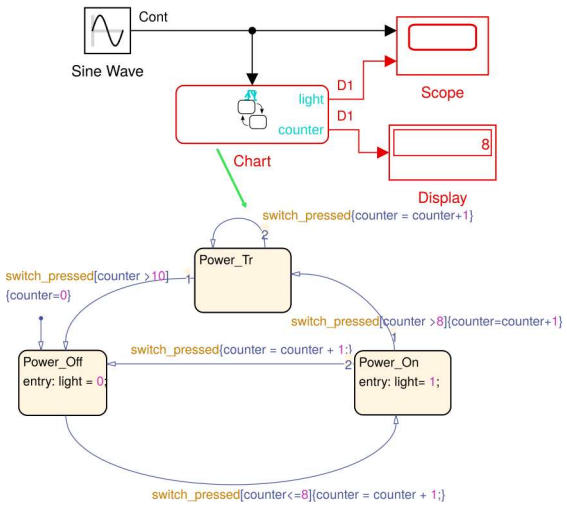


Figure 1: Example Stateflow Model

```
/* Chart: '<Root>/Chart' incorporates:
 * TriggerPort: '<S1>/switch_pressed'
 */
zcEvent = rt_ZCFcn(ANY_ZERO_CROSSING,
  &rtPrevZCX.Chart_Trig_ZCE,(rtDW.SineWave));
if (zcEvent != NO_ZCEVENT) {
  /* Gateway: Chart */
  /* Event: '<S1>:6' */
  /* During: Chart */
  switch(rtDW.bitsForTID1.is_c3_Switch_on_off_ext) {
    case IN_Power_Off:
      rtDW.light = 0.0;

      /* During 'Power_Off': '<S1>:1' */
      if (rtDW.counter <= 8.0) {
        /* Transition: '<S1>:9' */
        rtDW.counter++;
        rtDW.bitsForTID1.is_c3_Switch_on_off_ext =
          IN_Power_On;
        /* Entry 'Power_On': '<S1>:3' */
        rtDW.light = 1.0;
      }
      break;
  }
}
```

Listing 1: Translated Generated C-Code

MathWorks' documentation defines Stateflow as an environment for modeling and simulating combinatorial and sequential decision logic based on state machines and flow charts. Stateflow combines graphical and tabular representations, including state transition diagrams, flow charts, state transition-tables, and truth tables, to model how a system reacts to events, time-based conditions, and external input signals [6].

Matlab and Simulink functions can also be embedded in Stateflow diagrams, and interact with other blocks within the model via input and output connections, parameters and data. Stateflow is deterministic, thus ensuring the execution of transitions in an exact order. Figure 1 below depicts an example Stateflow model with Simulink blocks.

Simulink Coder and Embedded Coder: The Simulink Coder generates C/C++ source code for the Simulink or Stateflow models for real-time and non-real-time applications.

Matlab's code generators provide configuration options to include traceability between the input model and the generated code; by inserting tracecability tags (code-to-model) to Simulink blocks, Stateflow objects, and Matlab functions as comment above the line of the generated code. The Embedded Coder extends the Matlab-, Simulink Coder with features for generating compact code for embedded systems. Listing 1 depicts a snippet of the generated code for the Stateflow model in Figure 1.

Details on creating a Simulink or Stateflow model and the code generation process are out of the scope of this paper. For more information refer to [9].

4 OVERALL DEBUGGING CONCEPT

A prerequisite for debugging any binary code running on a target is building the source code with the `-g` compiler option (requests the compiler to generate debug information). The debug information is the representation of the relationship or mapping between the executable program instructions and its original source code – usually encoded in a predefined format. Most widely used is the DWARF format for ELF binaries. When debugging an executable, the debugger uses the debug information to map memory addresses to symbols, and the program counter value to source code locations and vice-versa, e.g. to support source line stepping. The debug information also helps the debugger to read out sections of the executable in order to retrieve the addresses of functions, variables (type, scope, location) etc.

We utilized the concept of source-level debugging for realizing model-level debugging. In analogy to the source-level debug information, our concept proposes the generation of a model-level debug information which is a mapping between model elements (properties, functions) and the generated source code. The debug information is provided to the debugger (LLDB) via extension plugins. The traceability tags embedded in the source code by the source code generator are used for the generation of debug information for an input model. How well an executable can be debugged at the

model-level depends on how detailed the traceability tags added to the code are.

In source-level debug information, there is a full mapping between lines of code and the machine code addresses. This information is used by the debugger for bi-directional mapping in order to set breakpoints at a source line or to return the source line corresponding to an instruction which causes a segmentation fault. Similarly, the debug information saved for each model element, as shown in Figure 2, enables a bi-directional mapping between source line and the model element. The source line is then mapped to the machine code address using the source-level debug information.

For example, to set breakpoints for /chart/, /state/ or /transition/-related points, the file name and line number referencing these points and their transitions are are stored as debugging information in the model object. The value of input- and output data (signals) for Simulink blocks can also be acquired by saving the variable name representing these signals and blocks in the source code as debugging information. Using the saved variable name the value of a model level signal can be read by the debugger when requested.

The Model level debugging is basically a wrapper on the source level debugger, as each model level command can be internally mapped to one or more source level debug commands. Details on how the debugging information is generated and how the commands are internally mapped to source level commands are explained in the subsequent paragraphs. In this section, the various steps of the proposed concept are described including their use of the background technologies XML Parsing, Clang and LLDB and their way of achieving the model level debugging of source code running on a target system.

In the following we provide a structural overview of our approach that describes the different modules of the architecture. It mainly consists out of the Matlab/Simulink components and the LLDB-Model-Plugin extension, as depicted in Figure 2.

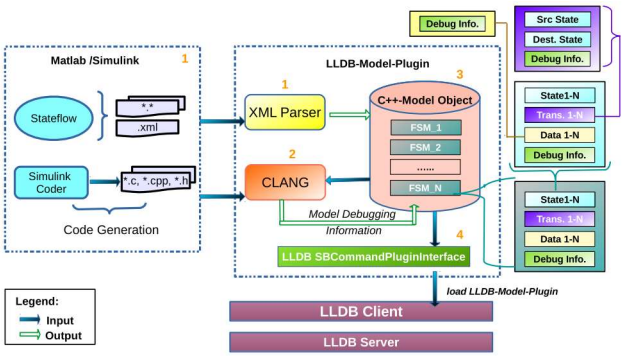


Figure 2: Model Aware Debugging - Structural Overview

Matlab/Simulink: These components represent the tools used for modeling the software and generating the production code. The **Stateflow Model** description (detailed properties of elements and mutual relationship) is saved as an XML document. The **Simulink Coder** generates C/C++ source code from the Simulink or Stateflow models for real-time and non-real-time applications with code-model traceability tags as comments above lines of the generated code.

LLDB-Model-Plugin: The LLDB-Model-Plugin is an extension of the LLDB for model aware debugging. It is responsible for generating model debug information from an input model's description XML file and its generated source code. It is implemented as a shared library, defines new model related debug commands for LLDB, and can be loaded by LLDB dynamically at run-time.

The components of the LLDB-Model-Plugin are described below:

XML Parser. The C++ Xerces XML parser is used by the LLDB-Model-Plugin to parse the model description file and create a C++ ModelObject that maintains all model related information.

Clang. As mentioned in Sec. 3.1, the Clang compiler is able to parse source code, check for errors and generate a language specific *abstract syntax tree (AST)*. The LLDB-Model-Plugin uses Clang to parse the AST of the generated source code and extract comments related to model elements as well as their source location (file name and line number) from the AST. The extracted information is then used in generating the debug information for the input model.

C++ ModelObject. This is an identical representation of the model description XML file as a C++ Class Object. For each model element, the name, unique identifier, and attributes relevant for debugging are captured in the model object. It extends the model description by maintaining debugging information for each model element. As shown in Figure 2 above, the C++ ModelObject maintains a list of all Stateflow charts $(\text{FSM}_1, ..., \text{FSM}_N)$. Each FSM object maintains lists of states, data signals, transactions and debugging information. Every state object maintains lists of sub-states, data signals, transactions and debugging information. A Transaction object maintains besides the debugging information, the source state and destination states; where as only the debugging information is saved for the data signal objects.

LLDB SBCommandPluginInterface. This C++ interface is provided as part of LLDB's C++ API for implementing C++ plugins for LLDB. Every newly defined lldb command must implement this interface which is instantiated when the command is added to the LLDB interpreter (`SBInterpreter`).

LLDB-Model-Plugin provides two views of debugging: a Forward View and a Backward View. The *Forward View* describes the extension of LLDB with model related commands, the handling of these commands and their usage by the user. The *Backward View* describes how the debugger internal state is communicated to the LLDB-Model-Plugin

for graphical representation to users. This paper however focuses on the forward view only.

5 IMPLEMENTATION WITH CLANG AND LLDB

This section describes the prototype implementation of the concept presented in Sec. 4 for debugging a Stateflow model with LLDB. The Stateflow model and source code were generated with Matlab/Simulink 2017b and the plugin implementation depends on Clang 6.0, LLDB 6.0 and the Xerces C++ 3.2 libraries. This paper focuses on a Stateflow model with generated C code for a generic real-time target.

5.1 Model to Code Mapping

The first step to map the code to the model is parsing the XML model description file. A C++ `ModelObject` is instantiated and filled with model elements while parsing the description file. When parsing the XML file, for any found DOM Element node (represents a Model element) an FSM, a State, a Transition or a Data object is instantiated and added to the `ModelObject`.

In the next step, the populated `ModelObject` instances and the source code are passed to Clang to generate an AST for the source file. Next, the AST is parsed to extract lines of codes for the source files and comments. The extracted information is used to generate debug information. Our methodology for mapping the entire source code to the model elements is to create an association between every line of code in the model source file and a traceability tag (comment) inserted by the generator.

5.1.1 Clang AST Parsing – Extracting Comments and Source Location Information. To traverse the full AST, we used the algorithm provided by Clang's `RecursiveASTVisitor`. When traversing the AST, we faced some challenges: Clang provides an option to parse all comments in the source code when constructing the AST, but only attaches comments to declaration (type `Decl`) nodes in the AST. It however provides an API to retrieve all comments in the source code from the ASTContext, but only after constructing the `TranslationUnit`. This makes it impossible to create an association between statements and comments in a single iteration. We resolved this issue by traversing the AST twice. With the help of two tables we therefore need to link comments to statement source locations. Line numbers are used as unique key in both tables.

I In the first iteration, the *statement table* with line number and an empty string as comment is filled. Once the translation unit is created, comments are read and the comment table is filled with line number and comment string. The second iteration updates every value entry (previously saved as an empty string) in the statement table with an associated comment from the comment table.

In order to ensure a correct association of statements and comments, we performed a source-to-source transformation of the generated code. This ensures that if-else and switch-case statements are all enclosed as compound statements (see Listing 2 and 3). This enabled us to easily map statements without any preceding comment within compound statements (scope of if, else, switch-case statements is taken into consideration), thereby avoiding wrongly generated debug information.

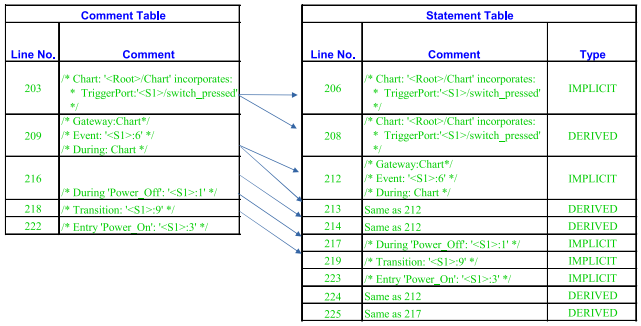


Figure 3: Statement/Comment Mapping

5.1.2 Generation of Model Debug Information. As mentioned above, the model debug information is mandatory for model aware debugging. Figure 3 depicts an example snippet of a comment and statement table for the code in Listing 1 for generating the debug information. Considering this example source code, keywords like *"Entry:", "Entry 'Statename':", "Transition:", "Event:"* etc. related to model elements are searched within each comment in the statement table and if found, the remaining information in a comment is processed to map the source line to model elements.

We describe a debug information entry implemented as a C++ Class `ModelDebugInfo` as follows:

- `ElementName` specifies the exact name of the model element for which the debug information was created.
- `ModelElementDebugType` specifies the model element type (chart, state, transaction, data etc.) for which the debugging information is created.
- `ModelDebugPoint` is the actual debug information consisting of a source name and a line number referencing the location in the source file where the model element is referenced.
- `DebugPointType` specifies if a debug point is an IMPLICIT or a DERIVED debug point. Implicit debug points refer to lines of code that are directly preceded by comments (e.g. line numbers 206, 212, 217 etc. of Listing 1); while lines of code that are not preceded by comments, but are associated to the last found comments within a semantically correct C++ source code scope (if/else, switch-case statements, functions etc.) are named DERIVED debug points (e.g. 208, 213, 214 etc.). See example in Figure 3.

5.2 LLDB Extensions

LLDB provides plug-in support for functionality and extensions. The plugin interface is available as part of

```
208     switch (rtDW.bitsForTID1.is_c3_Switch_on_off_ext) {
          case IN_Power_Off:
210         rtDW.light = 0.0;

212         /* During 'Power_Off': '<S1>:1' */
            if (rtDW.counter <= 8.0) {
214           /* Transition: '<S1>:9' */
              rtDW.counter++;
216           rtDW.bitsForTID1.is_c3_Switch_on_off_ext =
                IN_Power_On;
218           /* Entry 'Power_On': '<S1>:3' */
              rtDW.light = 1.0;
220         }
            break;
222
          case IN_Power_On:
224
```

Listing 2: Transformed Generated C-Code

```
switch (rtDW.bitsForTID1.is_c3_Switch_on_off_ext) {
  case IN_Power_Off: {
    rtDW.light = 0.0;

    /* During 'Power_Off': '<S1>:1' */
    if (rtDW.counter <= 8.0) {
      /* Transition: '<S1>:9' */
      rtDW.counter++;
      rtDW.bitsForTID1.is_c3_Switch_on_off_ext =
        IN_Power_On;
      /* Entry 'Power_On': '<S1>:3' */
      rtDW.light = 1.0;
    }
    break;
  }
  case IN_Power_On: {
```

Listing 3: Translated Generated C-Code

the LLDB debugger API and provided as a shared library. Listing 4 depicts the `lldb::PluginInitialize(lldb::SBDebugger dbg)` to be called within each LLDB plugin and a `SBCommandPluginInterface` which was implemented for every new model related command. The commands can be added to LLDB's `SBDebugger` by calling its interpreter object to a new command which can be either single- or multi-word. The debug information is accessible to the debugger by passing a reference to the C++ model object via each command.

```
class SBCommandPluginInterface {
public:
  virtual ~SBCommandPluginInterface() = default;
  virtual bool DoExecute(
    lldb::SBDebugger /*debugger*/,
    char ** /*command*/,
    lldb::SBCommandReturnObject & /*result*/) {
      return false;
  }
};
bool lldb::PluginInitialize(lldb::SBDebugger dbg);
```

Listing 4: LLDB Plugin Interface

```
lldb::SBCommand AddMultiwordCommand(const char *name,
                                    const char *help = nullptr);

lldb::SBCommand AddCommand(const char *name,
                           lldb::SBCommandPluginInterface *impl,
                           const char *help = nullptr);

lldb::SBCommand AddCommand(const char *name,
                           lldb::SBCommandPluginInterface *impl,
                           const char *help,
                           const char *syntax);

lldb::ReturnStatus HandleCommand(const char *command_line,
                                 lldb::SBCommandReturnObject &result,
                                 bool add_to_history = false);

lldb::ReturnStatus HandleCommand(const char *command_line,
                                 SBExecutionContext &exe_ctx,
                                 SBCommandReturnObject &result,
                                 bool add_to_history = false);
```

Listing 5: LLDB Command API

Our LLDB extension currently implements only the *Forward View Debugging* mode, which we consider as the handling of user input commands for model related debugging. For this view the `DoExecute` function as shown in Listing 4 is implemented for each model command. The first step is the parsing of the command line options to ensure that these correspond to the options specified when the commands were added to the command interpreter. Once the model element is determined from the command line argument, the command is handled appropriately. Most model aware debugging commands were internally mapped within the `DoExecute` function to a source level lldb command. For example to set a breakpoint for a state model element, the model aware command is internally mapped to setting source level breakpoints for source lines. The source file name and line number passed to the debugger are retrieved from the saved model debug information for that state element. With a single command several breakpoints can be set. Model related breakpoints will only be set for debug points with the `DebugPointType` IMPLICIT as DERIVED debug points shall be used only for mapping instructions (determining which model element is active for the current program counter address).

6 EXPERIMENTAL RESULTS

In order to experimentally evaluate our concept, we investigated the debugging of two state flow models from the MathWorks examples for source code generated in C and C++ programing language. Our experimental results are presented for the Stateflow example `sf_cdplayer.xls` which models a media player by using enumerated data in three Stateflow charts as shown in the figure below.

The core of the logic for controlling the CD Player/Radio is in the `CdPlayerModeManager` chart. This chart receives user inputs, such as whether a disc has been inserted and the choice for the radio mode (FM, AM, or CD). Then the chart determines the mechanical command to output. For example, this chart is responsible for making sure that a Rewind command is not issued in the absence of a disk in the player. The data types of input data `RadioReqMode` and `CdReqMode` as well as the output data `CurrentRadioMode` and `MechCmd` are defined as enumerated data types. The enumeration strings such as CD or FM are accessed directly in the chart for comparisons and assignments.

In the Matlab environment, the inputs to `CdPlayerModeManager` are provided by the chart `UserRequestChart` which is woken up at a periodic rate by Simulink and calls an external Matlab file named

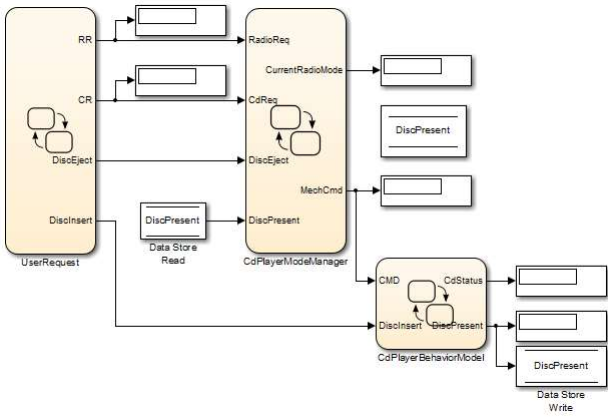


Figure 4: LLDB Command PluginInterface

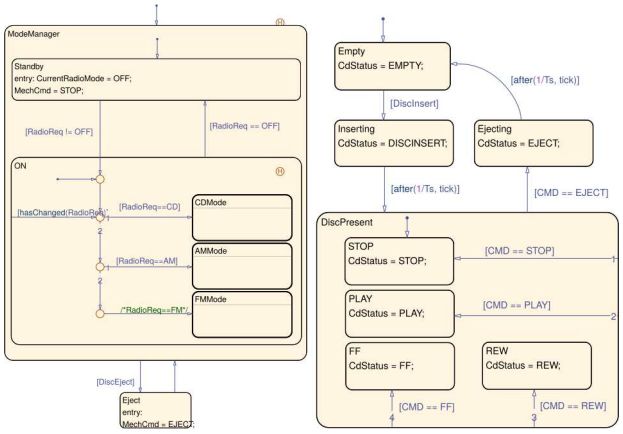


Figure 5: CD Behavior and Mode Manager Chart

`sfcdplayerhelper.m`. This Matlab file acts as an intermediary between the *Handle Graphics* panel and Simulink. For our experiments however, the input `UserRequest` was not included and therefore not all states could be reached. In a fully configured HIL setup, such a model is typically executed on a HIL simulator (host computer) and attached to the cross-compiled implementation on the target board via appropriate interfaces. Even without this refined environment model, basic testing of our concept was possible with the chart `UserRequestChart`.

The following sub-sections present the experimental result of the Forward-View debugging for the Media Player given above based on the concept in Sec. 4. This includes commands from the user to browse the model, setting breakpoints for model elements and observing where the debugger hits the breakpoint in the source code.

6.1 Loading the Plugin in lldb command line and start a binary (X86 Target)

Listing 6 shows the LLDB command line option to load a plugin.

```
user@host:~$ lldb
(lldb)plugin load /home/user/Model_LLDB_Debugger/libModel_LLDB_Debugger.so
  created Model & filled elements with Debug Information !
  All Model Commands successfully initialised !
(lldb)
```

Listing 6: Load LLDB Plugin

6.2 Browsing the Model

With the command `mprint` the user can browse the input model for detailed information as shown in Listing 7 below.

```
(lldb) mprint
Printing ModelInfo....
Model Name:sf_cdplayer_STT.slx Charts:2

Chart:
Name:CdPlayerModeManager ID:137  States:2
Name: CdPlayerBehaviorModel ID:254  States: 4

For more information call 'mprint -c <ChartName>'
For more information call 'mprint -s <stateName>-t <TransitionID>'

(lldb) mprint -s ModeManager
  ID: 56  Name: ModeManager  Parent: CdPlayerModeManager

SubStates:
   ID: 57  Name: Standby Parent: ModeManager
   ID: 58  Name: ON  Parent: ModeManager

Transitions:
 ID: 69  Parent: ModeManager Src: 0  Dest: 57
 ID: 68  Parent: ModeManager Src: 57 Dest: 58
 ID: 71  Parent: ModeManager Src: 58 Dest: 57

(lldb) mprint -s ON
State : ID :58  Name :ON  Parent : ModeManager
SubStates:
  ID: 59  Name: CDMode  Parent: ON
  ID: 61  Name: AMMode  Parent: ON
  ID: 62  Name: FMMode  Parent: ON
Transitions:
  ID: 133 Parent: ON  Src: 0  Dest: 74
  ID: 75  Parent: ON  Src: 58   Dest: 74
  ID: 73  Parent: ON  Src: 74   Dest: 59

(lldb)
```

Listing 7: Browsing with `mprint` command

6.3 Setting Breakpoints on a Model Element

```
(lldb) mbreakpoint set -s CDMode -sbrType en
  breakpoint set -f sf_cdplayer_STT.c -l 84
  breakpoint set -f sf_cdplayer_STT.c -l 92
  breakpoint set -f sf_cdplayer_STT.c -l 139
                 ......
  breakpoint set -f sf_cdplayer_STT.c -l 320

  Exiting set breakpoint
  Breakpoint 6: where = SF_CDPlayer`sf_cdplayer_STT_ModeManager + 682 at
      sf_cdplayer_STT.c:320, address = 0x000000000040bcba

  (lldb) mbreakpoint set -s ON
  breakpoint set -f sf_cdplayer_STT.c -l 64
  breakpoint set -f sf_cdplayer_STT.c -l 249
  breakpoint set -f sf_cdplayer_STT.c -l 215
                 .......
  breakpoint set -f sf_cdplayer_STT.c -l 376

  Exiting set breakpoint
  Breakpoint 13: where = SF_CDPlayer`sf_cdplayer_STT_ModeManager + 1003 at
      sf_cdplayer_STT.c:376, address = 0x000000000040bdfb

  Exiting set breakpoint
  (lldb)
```

Listing 8: Breakpoint command `mbreakpoint`

The command `mbreakpoint set -s <state> -sbrType <en=Entry, du=During, ex=Exit> -t <transition-id>` (aliased `mb`) provides the user with options of setting breakpoints on Model elements; and to enable, disable and delete existing breakpoints. It also provides an option to list all model related breakpoints set in the debugger. In order to verify our implementation, we set breakpoints (see Listing 8) on the entry point of the CDMode state by specifying the option `-sbrType en`. For the state ON the `-sbrType` option is not specified. In this case the breakpoints are set for all *Entry*, *During*, and *Exit* points of the state.

In order to evaluate if the breakpoint is currently set, we executed the run command to run the process of the executable previously started on the target. As can be seen in Listing 9 below, the process stopped at line 64, and 139, which correspond to the entry point of the states CDMode and ON as indicated by the commands above the lines in the code respectively.

```
(lldb) run
Process 9900 launched: '/home/user/SF_CDPlayer/SF_CDPlayer' (x86_64)
; ...
* thread #1, name = 'SF_CDPlayer', stop reason = breakpoint 7.1
frame #0: 0x000000000040c5f4 SF_CDPlayer`sf_cdplayer_S_enter_internal_ON at
       sf_cdplayer_STT.c:64
   61   static void sf_cdplayer_S_enter_internal_ON(void)
   62   {
   63     /* Entry Internal 'ON': '<S2>:58' */
-> 64     switch (sf_cdplayer_STT_DW.bitsForTID1.was_ON) {
   65      case sf_cdplayer_STT_IN_AMMode: {
   66       sf_cdplayer_STT_DW.bitsForTID1.is_ON =
      sf_cdplayer_STT_IN_AMMode;
   67       sf_cdplayer_STT_DW.bitsForTID1.was_ON =
      sf_cdplayer_STT_IN_AMMode;
(lldb) c
Process 9900 resuming
Process 9900 stopped
* thread #1, name = 'SF_CDPlayer', stop reason = breakpoint 3.1
    frame #0: 0x000000000040c7ce SF_CDPlayer`
      sf_cdplayer_S_enter_internal_ON at sf_cdplayer_STT.c:139
   136
   137        /* Entry 'CDMode': '<S2>:59' */
   138        /* '<S2>:59:1' CurrentRadioMode = RadioRequestMode.CD; */
-> 139        sf_cdplayer_STT_B.CurrentRadioMode = CD;
   140
   141        /* '<S2>:59:2' MechCmd =  CdRequestMode.STOP; */
   142        sf_cdplayer_STT_B.MechCmd = STOP;
```

Listing 9: Model Aware Debugging – Breakpoint hit

7 CONCLUSION AND FUTURE WORK

We have presented our approach towards model-aware debugging for model-based software development: a platform independent concept, leveraging existing open-source components like Xerces C++ and Clang. Based on the approach, we generated model-level debugging information from the source code embedded with traceability tags generated for a Stateflow model. The LLDB was extended with the model-level debugging information. Thus enabling a model-aware debugging of software applications.

With our approach several breakpoints can be easily set in a code by just specifying a model element, whereas a developer would have needed to go through the complete code; retrieve source locations for model elements and set break points for each location manually. This is however automated by our concept. Furthermore, our solution is an open-source-based and is applicable for software models of various types as long as there exist code-to-model traceability tags for the input model.

We believe therefore that, this approach is well suited for debugging at the model level because the software can be debugged on a real target system without source code instrumentation. One limitation however, is strict dependency of the tool on the traceability tags inserted by the source code generator and the XML model description file of Matlab/Simulink. For compatibility, the tool must be qualified for usage only with input files generated by a certain Matlab/Simulink version.

In the future, we intend to provide a possibility for the graphical representation of the model elements during debugging.

ACKNOWLEDGMENTS

The research leading to these results was carried out within the COMPACT (Cost Efficient Smart System Software Synthesis) ITEA3 project and was funded by the German Ministry of Education and Research (BMBF) under grant agreement No. 01IS17028H.

REFERENCES

[1] María del Mar Gallardo, Pedro Merino, and Ernesto Pimentel. 2002. Debugging UML Designs with Model Checking. *Journal of Object Technology* 1, 2 (July 2002), 101–117. https://doi.org/10.5381/jot.2002.1.2.a1

[2] Eran Gery, David Harel, and Eldad Palachi. 2002. Rhapsody: A Complete Life-Cycle Model-Based Development System. In *Integrated Formal Methods*, Michael Butler, Luigia Petre, and Kaisa Sere (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–10.

[3] Liang Guo and Abhik Roychoudhury. 2008. Debugging Statecharts Via Model-Code Traceability. In *Leveraging Applications of Formal Methods, Verification and Validation*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 292–306.

[4] W. Haberl, M. Herrmannsdoerfer, J. Birke, and U. Baumgarten. 2010. Model-Level Debugging of Embedded Real-Time Systems. In *10th IEEE International Conference on Computer and Information Technology*. 1887–1894. https://doi.org/10.1109/CIT.2010.323

[5] Markus Herrmannsdoerfer, Wolfgang Haberl, and Uwe Baumgarten. 2009. Model-level Simulation for COLA. In *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering (MISE '09)*. IEEE Computer Society, Washington, DC, USA, 38–43. https://doi.org/10.1109/MISE.2009.5069895

[6] MathWorks. 2018. Stateflow Documentation - MathWorks Deutschland. Retrieved October 11, 2018 from https://de.mathworks.com/help/stateflow/

[7] MathWorks. 2018. Test and Debug Simulations - Matlab & Simulink - MathWorks Deutschland. Retrieved October 16, 2018 from https://de.mathworks.com/help/simulink/test-and-debug-simulations.html

[8] MathWorks. 2018. Test Models in Real Time - Matlab & Simulink - MathWorks Deutschland. Retrieved October 16, 2018 from https://de.mathworks.com/help/sltest/ug/test-models-in-real-time-and-assess-results.html

[9] MathWorks. 2018. Trace Stateflow Elements in Generated Code - Matlab & Simulink - MathWorks Deutschland. Retrieved October 11, 2018 from https://de.mathworks.com/help/ecoder/ug/trace-stateflow-elements-in-generated-code.html

[10] LLVM Projects. 2018. Clang C Language Family Frontend for LLVM. Retrieved October 29, 2018 from https://clang.llvm.org/

[11] LLVM Projects. 2018. LLDB Homepage. Retrieved October 11, 2018 from https://lldb.llvm.org/

Fast SystemC Processor Models with Unicorn

Lukas Jünger, Jan Henrik Weinstock, Rainer Leupers and Gerd Ascheid

Fast SystemC Processor Models with Unicorn

Lukas Jünger
RWTH Aachen University
juenger@ice.rwth-aachen.de

Jan Henrik Weinstock
RWTH Aachen University
weinstock@ice.rwth-aachen.de

Rainer Leupers
RWTH Aachen University
leupers@ice.rwth-aachen.de

Gerd Ascheid
RWTH Aachen University
ascheid@ice.rwth-aachen.de

ABSTRACT

In this work a Virtual Platform (VP) is presented containing a novel processor model for the latest ARMv8 instruction set architecture. This processor model was constructed using the Unicorn emulator [6]. The necessary modifications to the Unicorn emulator and subsequent performance improvements during SystemC simulation are shown in detail. In addition the integration into a VP using a state-of-the-art SystemC modeling library is described. A comparison is made with a VP containing another similar processor model, highlighting the benefits of using Unicorn for processor modeling in a SystemC environment.

CCS CONCEPTS

• **Hardware** → **Hardware-software codesign**; • **Computing methodologies** → *Discrete-event simulation*;

KEYWORDS

Electronic System Level, QEMU, SystemC

ACM Reference Format:
Lukas Jünger, Jan Henrik Weinstock, Rainer Leupers, and Gerd Ascheid. 2019. Fast SystemC Processor Models with Unicorn. In *Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '19), January 21–23, 2019, Valencia, Spain.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3300189.3300191

1 INTRODUCTION

In the fast-paced development cycles of today, Virtual Platforms (VPs) have become an essential tool in embedded software development. VPs are typically available long before their hardware counterparts, allowing to start software development earlier. They enable hardware/software co-design, since insights gained from early software development can be of use to the hardware designers. Even though VPs do not model every detail of real hardware, having more time for software development and testing yields more mature and stable software. This, in turn, speeds up the bringup on the physical hardware of the final product once it is available.

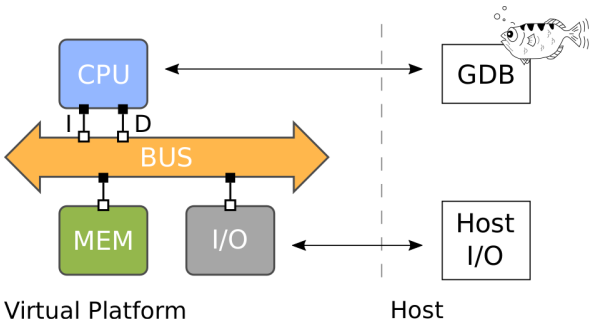


Figure 1: Overview of example VP

SystemC Transaction-Level Modeling 2.0 [1] (TLM) has become the de facto standard for constructing VPs. It allows for the VP to be described using C++, which is then compiled into a fast simulation executable. Because the VP is a C++ program, it can be inspected using off-the-shelf C++ debuggers and other analysis tools. Hence, VPs offer a high degree of introspection, making them ideal for debugging, and their malleability and interoperability allows users to customize them to their needs. An overview of an exemplary VP intended for software development is given in Figure 1. It consists of a processor model with debugger integration, memory and an Input/Output (I/O) device, which is connected to host I/O.

Ideally, there should be no discernible difference between using a VP or physical hardware from the point-of-view of the software developer. Notably, it should react to inputs as quickly as physical hardware. Therefore, rapid simulation is a key productivity feature of a VP. A VP usually consists of multiple component models such as processors, buses, memories and peripherals. Since all of these components are simulated, they have an effect on the overall simulation performance. The processor model is a core component of the VP and normally contains an Instruction Set Simulator (ISS) for the simulated target Instruction Set Architecture (ISA). During simulation a substantial amount of execution time is spent in the ISS of the processor model. Therefore, a high performance ISS is paramount to achieving rapid simulation

RAPIDO '19, January 21–23, 2019, Valencia, Spain

ACM ISBN 978-1-4503-6260-3/19/01…$15.00
https://doi.org/10.1145/3300189.3300191

speed, yielding smooth VP operation in conventional software development environments.

In this work, a TLM processor model using the open-source Unicorn emulator [6] is introduced. ARMv8 was selected as the target ISA, because it is the most common 64-bit embedded systems ISA. This processor model was subsequently used to construct a full VP. Different benchmarks were evaluated on the VP to assess the performance of the Unicorn-based processor model.

The remainder of this work is structured as follows. In Section 2, work related to the topic is surveyed and summarized. Afterwards, Section 3 describes the Unicorn TLM wrapper and GNU Debugger (GDB) integration. The VP, that was built using the new Unicorn-based processor model, is described in Section 4. An experimental evaluation of the VP with industry-standard benchmarks is given in Section 5. Finally, a conclusion is drawn in Section 6.

The contributions of this work are:

- Design of a novel ARMv8 SystemC processor model using the Unicorn emulator
- A new instruction counting mechanism for the Unicorn emulator, suitable for rapid SystemC simulation
- Construction of a realistic VP using the novel processor model
- Assessment of the performance of the Unicorn-based VP in comparison with a similar state-of-the-art VP

2 RELATED WORK

Generally, an ISS is needed when building a processor model from scratch. ARM offers commercial processor models, that are equipped with a SystemC interface: the ARM FastModels [2]. Besides the SystemC interface, these models offer different debug and scripting Application Programming Interfaces (APIs).

Another option is the open-source QEMU system emulator [3]. QEMU contains a Dynamic Binary Translation (DBT) ISS component for many different ISAs.

The main idea of DBT is to translate binary instructions from the target to the host instruction set at runtime. QEMU also stores and then re-uses the translated instructions. This way, the target instruction fetch and decode is only executed once as opposed to an interpreting ISS. Different optimization techniques can be applied to speed up the DBT process. QEMU is not a stand-alone processor model. It also includes models of other hardware components, including, but not limited to, graphics card, sound card, USB controllers and memory controllers. By combining these components, QEMU is capable of simulating a complete system. In addition, it includes a GDB server, which allows the user to connect a GDB instance to debug the software running on the simulated system. However, QEMU does not have a SystemC interface and is geared towards full system emulation. It is not usable in a VP by itself, since it lacks the customizability and interoperability of SystemC.

To address the lack of a SystemC interface in QEMU, the QBOX platform emulation environment by GreenSocs [4] adds a SystemC wrapper to the emulator. With this, QEMU can be integrated into a TLM simulation. It also keeps the GDB server intact to enable debugging of the target software. Since QEMU is kept mostly unmodified, including the hardware component models, there is inherent overhead if only a processor model is needed in the VP. AMVP [9], selected as comparison in this work, uses QBOX in its processor model.

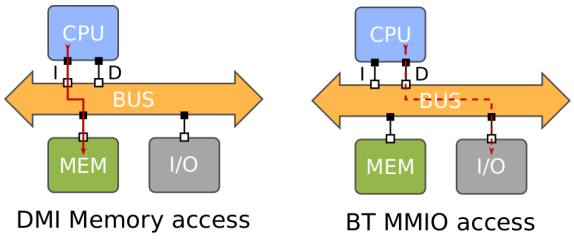


Figure 2: Different kinds of memory accesses

In 2015 Nguyen et al. presented their Unicorn emulator [6]. Unicorn is based on the QEMU system emulator version 2.2.1, but it keeps only a small subset of the functionality of QEMU. The fast DBT based ISSs are kept, while the other component models are removed. A lightweight API is provided to use the ISSs. Unicorn does not include a TLM compatible interface and thus cannot be used in a SystemC VP without modification. This work proposes a solution to this problem.

3 EMBEDDING UNICORN IN SYSTEMC

In this work, the Unicorn emulator [6] is used to build a TLM-compatible ARMv8 processor model. Since Unicorn does not include a TLM interface, some modifications are made to the existing code base. In this section, these modifications are described in more detail. The Unicorn-based processor model was constructed using the Virtual Components Modeling Library (VCML) [10]. VCML provides base classes, that simplify the construction of SystemC hardware components models. It also includes several complete SystemC models for assembling a VP, such as memory, bus and peripheral models.

From the perspective of VCML, Unicorn is a processor model. As such, mechanisms for memory access, Memory-Mapped I/O (MMIO) and interrupts had to be designed. These are described in Section 3.1. In addition, VCML includes a GDB Remote Serial Protocol (RSP) [7] server implementation, that simplifies the debugger integration, which is essential for a VP. Without it, the VP lacks core features for software debugging and development. Therefore, GDB support was added in the Unicorn-based processor model. More details on the GDB integration are provided in Section 3.3. Unicorn is not designed for speed and thus specific performance enhancements were added. These enhancements are described in Section 3.4.

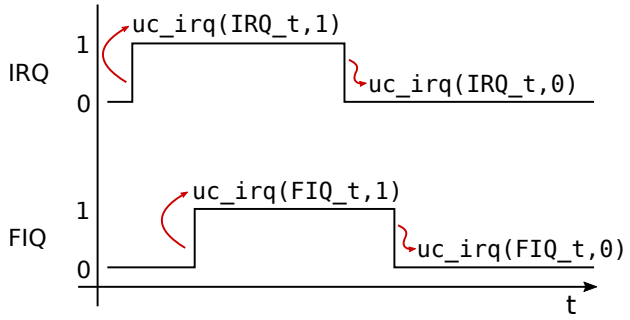


Figure 3: External interrupts triggering function calls to the processor model

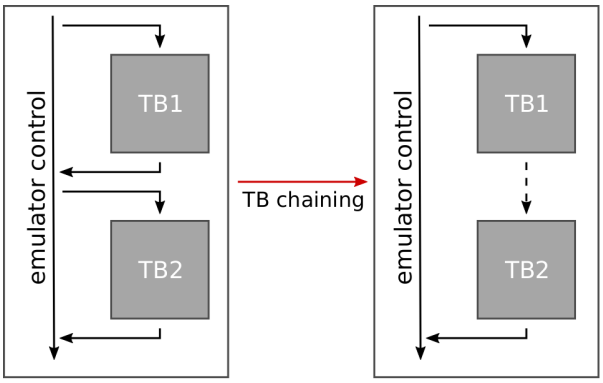


Figure 4: Execution flow before and after TB chaining optimization

3.1 Memory

For this work, two types of memory accesses have to be distinguished. The first type is a standard memory access from the processor model to the RAM model. The second type of memory access is MMIO, which happens when the processor model accesses a memory-mapped peripheral e.g. a UART. In the SystemC domain, these two kinds of accesses have to trigger different behaviors. When the processor model is accessing the RAM, the access can be conducted using the TLM Direct Memory Interface (DMI). In this case the memory access is done directly via a pointer, which is the fastest method for modeling this kind of behavior. In case the processor model is accessing a MMIO peripheral model, the memory access has to be conducted via the TLM Blocking Transport interface (BT). This is due to the fact, that the peripheral model usually has to react to the access, e.g., by printing a character on the screen, and thus needs to be informed of it. The two different types of memory accesses are depicted in Figure 2.

To the processor it is generally unknown whether a memory access targets the RAM or a MMIO peripheral. The distinction between the different memory access methods is therefore to be made by the TLM wrapper. The processor fetches instructions and data from the RAM via its corresponding data and instruction ports, which are connected to a bus model. Unicorn includes a mechanism that allows for mapping host memory directly to the emulator memory space via a pointer. This mechanism is used for mapping the RAM model memory to the emulator at the end of the SystemC elaboration phase. After the RAM is mapped, program execution can start.

Unicorn allows to connect callback functions to different memory events, such as an unmapped memory access. This callback function is executed when an instruction accesses memory that is not mapped in the address space of the emulator. When this happens, the emulator cannot continue execution and the following algorithm is executed to resolve the issue:

(1) Try to acquire a DMI pointer and map the missing memory into the memory space of the emulator.
(2) If a DMI pointer cannot be acquired, a BT is used for the memory access, and callback functions are registered on the address to handle the memory access via a BT in the future.

Using the method described above, all memory accesses occurring during the SystemC simulation can be handled.

3.2 Interrupts

ARMv8 processors feature the external IRQ and FIQ interrupt signals, which have to be included in the processor model to achieve full compatibility The processor model has to be able to react to changes in these signals. Unfortunately, the Unicorn API does not expose the interrupt functionality of the ISS. Therefore, the API had to be extended to expose this functionality to the TLM wrapper. Interrupts are modeled as `sc_signal<bool>`. The processor model reacts the signal changes by executing an IRQ trigger (`uc_irq()`) function on rising and falling signal edges. This is depicted in Figure 3.

In this IRQ trigger function, the corresponding QEMU function is called via the extended Unicorn API, initiating the CPU state changes to reflect the occurrence of the interrupt.

3.3 GNU Debugger Integration

In order to ease the bringup of the target software on the processor model, a debugger integration is indispensable. The VCML processor model base class includes an integration of GDB RSP. GDB can be connected to the running VP remotely via a network connection and does not need to be executed on the same machine.

For GDB support, the processor model needs to be able to read and write the emulated processor registers and set and remove breakpoints. Many of the ARMv8 registers are accessible via the Unicorn API and the missing ones were added for this work. Breakpoint support was added using Unicorn memory callback functions. When GDB sets a breakpoint on the processor model, a memory callback is added for the

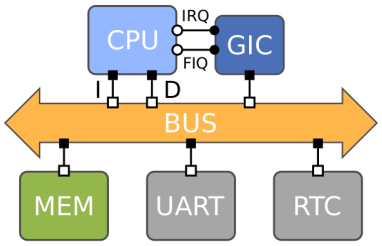


Figure 5: Overview of the Unicorn VP

corresponding memory address. Once execution reaches this address, the VCML GDB server stops execution and passes control to GDB to resume interactive debugging. Memory access is provided to GDB by the VCML GDB integration directly via the TLM data and instruction sockets, and therefore needs not to be implemented by the processor model.

3.4 Performance Optimization

TLM simulations use temporal decoupling to increase simulation performance. Following this concept, the processor model is allowed to run ahead of the global simulation time until the next synchronization point is reached. The largest amount of time a thread may differ from the global simulation time is referred to as the *quantum*. The size of the quantum corresponds to the amount of instructions the processor model may execute in between two synchronization points. In order to be able to execute a certain number of instructions, there needs to be an instruction counter and a way to exit the emulator when the desired number of instructions was executed. There are several ways in which the instruction counting can be implemented. In order to evaluate the different methods, more detailed knowledge of the Unicorn emulator is required.

As mentioned above Unicorn uses the DBT ISSs from the QEMU system emulator. The main component of these ISSs is the Tiny Code Generator (TCG), which handles the DBT. First, the target ARMv8 instructions are translated into TCG instructions, which are then optimized and translated to host instructions. The TCG translates on Translation Block (TB) granularity. A TB is a coherent block of instructions that can only be entered at its beginning and exited at its end, e.g., via a branch instruction. To further improve performance the TCG has a TB cache, that stores previously translated TBs for later reuse. At the end of a TB, execution can continue in two places, which correspond to whether the branch at the end of the TB was taken or not. This is used for an optimization referred to as *TB chaining*. When the end of a TB is reached, Unicorn checks which TB to execute next. It then stores the information on which TB followed due to which branch decision. When the TB is executed next time and the same branch decision is made, the execution directly continues at the correct TB without the need for another TB lookup. If at this point the other branch option is taken, this information is also stored together with the subsequent TB and when the TB is executed again, execution can continue directly

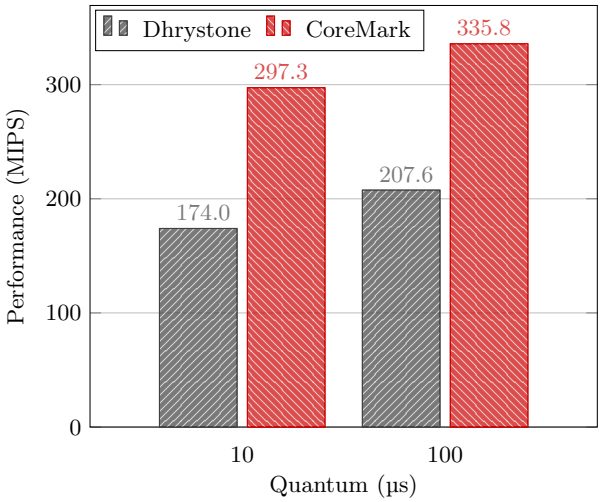


Figure 6: Benchmark results for Dhrystone and CoreMark on the Unicorn VP

for either branch decision. This optimization is depicted in Figure 4.

Unicorn implements instruction counting by branching to a counting routine after every instruction. This counting routine also checks whether execution should stop, in case the desired number of instructions was executed. Since Unicorn allows stopping execution at any address, the TB cache has to be flushed regularly. This is necessary, because Unicorn can also stop and continue execution somewhere inside a TB and not only at the beginning and end. In this case, at least the corresponding TB has to be re-translated. However, Unicorns default behavior is to flush the entire TB cache. When a precise instruction count is not needed, this behavior introduces considerable performance overhead, since it effectively degrades the TCG to an interpreter. In a TLM simulation a small TLM quantum overshoot is acceptable, meaning that execution can always continue until the end of the TB is reached. Therefore, it is sufficient to only count instructions on the TB level. This can be done efficiently by adding host machine instructions at the beginning of the TB to increase an instruction counter in host memory by the number of instructions in this TB. This also works when TBs are chained. Branching to a counting routine is not necessary. For this work, an instruction counting mechanism per TB, as described above, was added to the Unicorn emulator. The performance difference between the two methods is shown in Section 5.

The default behavior of Unicorn is still useful for debugging, where single instruction steps are common. Thus, the default behavior is enabled in the processor model when the debugger issues a single instructions step. When the debugger issues a continue command, the faster instruction counting mode is re-enabled.

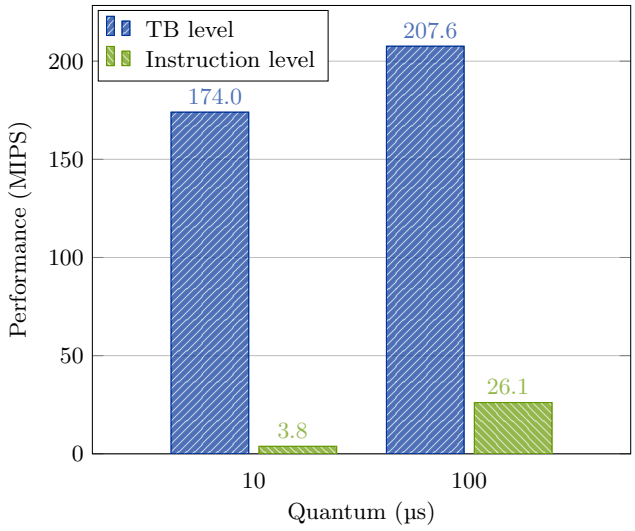


Figure 7: Benchmark results for Dhrystone on the Unicorn VP with TB and instruction granularity counting

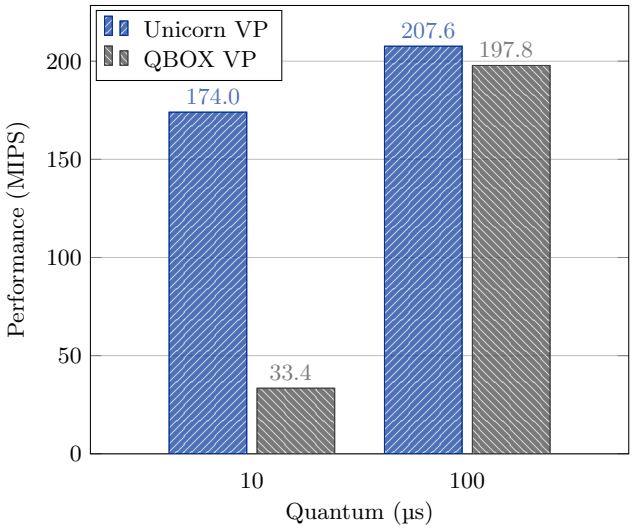


Figure 8: Benchmark results for Dhrystone on the QBOX and Unicorn VPs

4 THE UNICORN VIRTUAL PLATFORM

The Unicorn-based SystemC processor model, described in Section 3, was used to build a VP for experimental evaluation. This VP is described in this section. Figure 5 shows a platform overview of the complete VP.

Besides the processor model, the VP contains the following components:

- A generic bus model
- An ARM GICv2 interrupt controller model
- A generic memory model
- A Real-Time Clock (RTC) model
- A PL011 ARM PrimeCell UART peripheral model

All components besides the RTC are part of the VCML [10]. The RTC was implemented for this work. It is used to provide a reference time to the benchmark target software. The target software is copied to the memory before the SystemC simulation starts.

5 EXPERIMENTAL EVALUATION

To evaluate the performance of the processor model, described in Section 3, different benchmarks were executed on the Unicorn VP, introduced in Section 4. CoreMark [5] and Dhrystone [8] were ported to the VP as industry-standard benchmarks. The following results were acquired on an octa-core Intel i7-7700 host system with 32GB RAM, running CentOS 7.3.1611. Each experiment was repeated ten times and the results were averaged for the final result.

First, the CoreMark and Dhrystone benchmarks were executed on the Unicorn VP with set SystemC quanta of 10 µs and 100 µs. The processor model performance was measured in Million simulated Instructions Per wall clock Second (MIPS) using the profiling facilities of the VCML. The results are summarized in Figure 6.

It can be observed, that the CoreMark benchmark executes faster than the Dhrystone benchmark. This is likely caused by the fact, that Dhrystone is more memory intensive and the processor model has to leave the ISS more frequently to access the memory, which is costly.

As described in Section 3.4, the performance of Unicorn in a SystemC VP was improved by adding a novel instruction counting mechanism. The performance improvement was experimentally evaluated with the Dhrystone benchmark, which was executed on the Unicorn VP with the proposed TB-granularity and the original instruction granularity counting mechanism. The MIPS performance of the processor model was measured using the VCML profiling facilities. The results are visualized in Figure 7. Here it can be observed, that a significant performance improvement was achieved. However, one has to keep in mind that this comes at the cost of accuracy, since now TLM quantum overshoots must be tolerated. However, in a loosely-timed simulation this is generally not a problem.

In order to assess the processor model performance in comparison with similar models, a comparison with a single-core, sequential version of the QBOX-based [4] AMVP [9] was performed. The Dhrystone benchmark was executed on both VPs with two different TLM quanta of 10 µs and 100 µs. As before, the MIPS performance was measured using the VCML profiling facilities in both VPs. The results are summarized in Figure 8.

The Unicorn VP outperformed AMVP in both settings. It can be observed, that Unicorn was significantly faster than AMVP for the 10 µs quantum. From this it can be deduced, that the overhead of entering and leaving the QBOX emulator dominates the performance difference. For the 10 µs quantum

the ISS has to be paused and resumed more often, which is why the lightweight Unicorn-based processor model is significantly faster here. With a higher quantum, more time is spent inside the ISS. Since Unicorn and QBOX both use the QEMU TCG, the performance difference is not as significant.

6 CONCLUSION

This work shows, that Unicorn is a promising candidate for use as a TLM processor model. The missing TLM wrapper can be implemented using Unicorn callback functions and minor extensions to the Unicorn API. A big advantage is, that Unicorn is lightweight and only includes what is absolutely necessary for processor emulation. It keeps the QEMU DBT ISS with TCG and with some modifications can be brought to rapid simulation speed. The performance of the constructed VP is sufficient for interactive debug and higher than comparable VPs. When the VP is used as a debug target, no difference is observable between the VP and physical hardware.

REFERENCES

[1] IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, Jan 2012.

[2] ARM Holdings. Virtual Prototypes: Fast Models. https://developer.arm.com/products/system-design/fast-models, 2018. Online; accessed 21-10-2018.

[3] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[4] G. Delbergue, M. Burton, F. Konrad, B. Le Gal, and C. Jego. QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[5] Embedded Microprocessor Benchmark Consortium. CoreMark: An EEMBC Benchmark. http://www.eembc.org/coremark, 2018. Online; accessed 21-10-2018.

[6] A. Q. Nguyen and H. V. Dang. Unicorn: Next Generation CPU Emulator Framework. In *Proceedings of the 2015 Blackhat USA conference*, 2015.

[7] R. Stallman, R. Pesch, and S. Shebs. GDB Remote Serial Protocol. In *Debugging with GDB: the GNU Source-Level debugger, Version 5.1.1*, pages 281–292. Free Software Foundation, Boston, USA, 9 edition, 2002.

[8] R. P. Weicker. Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules. *SIGPLAN Not.*, 23(8):49–62, Aug. 1988.

[9] J. H. Weinstock, R. L. Bücs, F. Walbroel, R. Leupers, and G. Ascheid. AMVP - A High Performance Virtual Platform Using Parallel SystemC for Multicore ARM Architectures: Work-in-progress. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, CODES '18, pages 13:1–13:2, Piscataway, NJ, USA, 2018. IEEE Press.

[10] Weinstock, Jan Henrik. Virtual Components Modeling Library. https://github.com/janweinstock/vcml, 2018. Online; accessed 21-10-2018.

Fast Virtual Prototyping for Embedded Computing Systems Design and Exploration

Amir Charif, Gabriel Busnot, Rania Mameesh, Tanguy Sassolas and Nicolas Ventroux

Fast Virtual Prototyping for Embedded Computing Systems Design and Exploration

Amir Charif, Gabriel Busnot, Rania Mameesh, Tanguy Sassolas and Nicolas Ventroux
Computing and Design Environment Laboratory
CEA, LIST
Gif-sur-Yvette CEDEX, France
firstname.surname@cea.fr

ABSTRACT

Virtual Prototyping has been widely adopted as a cost-effective solution for early hardware and software co-validation. However, as systems grow in complexity and scale, both the time required to get to a correct virtual prototype, and the time required to run real software on it can quickly become unmanageable. This paper introduces a feature-rich integrated virtual prototyping solution, designed to meet industrial needs not only in terms of performance, but also in terms of ease, rapidity and automation of modelling and exploration. It introduces novel methods to leverage the QEMU dynamic binary translator and the abstraction levels offered by SystemC/TLM 2.0 to provide the best possible trade-offs between accuracy and performance at all steps of the design. The solution also ships with a dynamic platform composition infrastructure that makes it possible to model and explore a myriad of architectures using a compact high-level description. Results obtained simulating a RISC-V SMP architecture running the PARSEC benchmark suite reveal that simulation speed can range from 30 MIPS in accurate simulation mode to 220 MIPS in fast functional validation mode.

1 INTRODUCTION

To keep up with a fast-evolving and highly competitive embedded system industry, designers are compelled to deliver complete working solutions under tight delay and budget constraints. To make this possible, hardware architectures, as well as the full software stacks that drive them should be validated and optimized as early as possible in the design process. In this context, Virtual Prototyping has been widely adopted as a cost-effective solution for early hardware and software co-validation.

The rapid adoption of virtual prototyping solutions was greatly facilitated by the emergence of the SystemC/TLM 2.0 standard [1], which, in addition to offering interoperability and reusability of SystemC models, provides several abstraction levels to cope with varying needs in accuracy and speed. More recently, in response to an increasing demand for simulation speed, the use Dynamic Binary Translation (DBT) for CPU modelling has gained in relevance [17], [20], [10], and has effectively set a new standard for simulation performance in early prototypes.

However, as systems grow in complexity and scale, it is now more vital than ever that virtual prototyping solutions be able to wisely exploit these technologies to offer proper balance between simulation representativeness and execution speed throughout the design process. Moreover, at this level of complexity, the time required to model and explore new architectures can quickly become unmanageable, especially at the earliest stages, where it is often necessary to make heavy alterations before reaching a stable prototype.

To cope with these new difficulties, virtual prototyping solutions need to meet new requirements not only in terms of performance, but also in terms of ease, rapidity and automation of system modelling and design space exploration. This paper introduces VPSim, a feature-rich integrated virtual prototyping solution that addresses the aforementioned challenges based on three major contributions:

- A new way of integrating the QEMU emulator, making its rich CPU and peripheral model portfolio available as a collection of SystemC modules. Our method can leverage the technologies used in QEMU, such as Dynamic Binary Translation (DBT) and paravirtualization to bring outstanding simulation speeds into the more deterministic and standardized SystemC domain.
- An Accuracy Control framework, that can be used to define regions of interest in both the software and the simulated hardware dynamically. A system designer can therefore enable accurate simulation only on parts of the design and portions of code that are of interest, guaranteeing the best possible trade-off between performance and accuracy given specific needs.
- A generic and powerful infrastructure for dynamic platform composition and design space exploration (DSE). It allows a single compiled executable to be used to model an infinity of architectures. Combined with a Python scripting front-end, it makes it possible to simulate, explore and optimize highly complex systems in a single compact script.

RAPIDO '19, January 21–23, 2019, Valencia, Spain

ACM ISBN 978-1-4503-6260-3/19/01...$15.00
https://doi.org/10.1145/3300189.3300192

The remainder of this paper is organized as follows: In Section 2, we review some related tools and methods. Section 3 provides a complete high-level view of the features and capabilities of the VPSim tool. In Section 4, details on the underlying implementation challenges are presented. The performance of our tool is evaluated in Section 5, before concluding in Section 6.

2 RELATED WORKS

The first generation of Virtual Prorotyping (VP) solutions used functional Instruction Set Simulators (ISSs) with more or less internal low-level details such as pipeline stages. Among such solutions, GEM5 [7, 16] is a discrete-event simulator able to dynamically switch between different abstraction levels. Detailed CPU models with full pipelining description can be simulated at 0.1 Million Instructions Per Second (MIPS), whereas instruction-accurate CPU models can reach 1 MIPS. Other MPSoC modeling environments, such as SESAM [24] or Unisim [3] used instruction-based ISS generation librairies to support the modeling of various CPUs, reaching approximately 10 MIPS. However, this accuracy and ISA flexibility come at the cost of limited simulation speed, which hampers their capacity to address complex systems embedding several CPUs and running full-fledged OSes.

To address this complexity, a recent trend is the use of Dynamic Binary Translation (DBT). DBT consists in dynamically translating instructions of the modeled ISA (guest instructions) to host ones (usually x86), whenever needed during guest code execution, yielding very high execution speeds.

Today, all high-performance simulation environments use DBT. Industrial solutions are led by solutions like Virtualizer [25], Vista [19], VPS [18], VLAB [26], FastModels [20] or, for instance, Simics [22]. However, these come at a significant cost and do not offer the degree of customization that is required when designing new architectures. In addition, they do not provide fast design space exploration capabilities.

Amongst open-source virtual platforms, OVP [17] uses processor models simulated through a closed-source DBT engine named OVPSim reaching hundreds of MIPS. An OVP model can be wrapped for inclusion in a SystemC model using TLM 2.0 interfaces. However, OVPSim cannot provide performance evaluation as the models are timed relying on instruction count and neglecting memory access timings. In addition, OVP claims to support parallel multiprocessor simulation but this is at the cost of deterministic execution loss.

QEMU [4] is an open source DBT based emulator supporting many CPU models but does not provide performance estimation. Several works have tried to embed QEMU within a SystemC simulation environment to provide both determinism and timing evaluation [15]. In [15], the authors wrap QEMU processors within SystemC threads and investigate several QEMU instrumentation options to take into account instruction count and data access latency. Depending on whether synchronization shall occur on every data access or on a periodic basis, the simulation speed varies from 3 to 60 MIPS. However, the cumbersome annotation is to be performed for every ISA and possible accuracy settings are limited to a few predefined configurations. GreenSocs [10, 12, 13] propose a QEMU-based framework that exploits QEMU MMIO callback mechanisms to access external SystemC peripherals. This allows for very fast simulation for applications with few IO communications, as the execution mostly takes place in the context of QEMU. Unfortunately, this solution runs QEMU and SystemC in separate kernel threads, requiring frequent synchronization and precluding determinism.

The solution we propose integrates QEMU by executing its CPU and peripheral models in the context of SystemC threads, thereby preserving the predictability of Single-Threaded SystemC simulation. The accuracy of memory accesses, including instruction fetches, can be configured at a very fine granularity, and may change dynamically during simulation.

3 A USER-LEVEL VIEW OF VPSIM

VPSim is a key asset in charge of virtual prototyping and DSE within SESAM, an integrated EDA framework for complex electronic systems ranging from Cyber-Physical Systems to Microservers. SESAM provides a holistic environment to address HW/SW co-design, exploration and validation through Virtual Prototyping, HW prototyping and emulation while taking into account power, temperature and reliability factors. This section provides a high-level view of the major features and capabilities of VPSim as perceived by the end-user.

VPSim is a tool that was designed specifically to accelerate software/hardware co-validation at the earliest design stages of all kinds of computer architectures. It can be used to easily compose, simulate and explore new hardware architectures, but also to run, profile and debug full software stacks on the simulated platforms.

Figure 1 gives a global overview of how VPSim can be used. The user specifies a platform using a *platform composition front-end*, which forwards a *high-level platform description* to a central VPSim object named the *Platform Builder*. The Platform Builder uses this description, and a library of registered *components* to construct and run a SystemC simulation. Components can be either internal hardware models, or *Proxy Components* for interfacing with external subsystems. VPSim loads software binaries through the `ElfLoader` and `BlobLoader` special components. Once the SystemC simulation is launched, the user can use standard debugging tools or VPSim's built-in debug and profiling utilities, to evaluate and optimize their software. VPSim collects *fine-grained per-components statistics* and pushes them back to the high-level front-end, which may use them to present performance profiles to the user or to perform design space exploration.

VPSim is also capable of operating as merely one component of a wider Cyber-Physical System (CPS) simulation, in compliance with the Functional Mockup Interface (FMI) co-simulation standard [8]. In this mode, VPSim is packaged

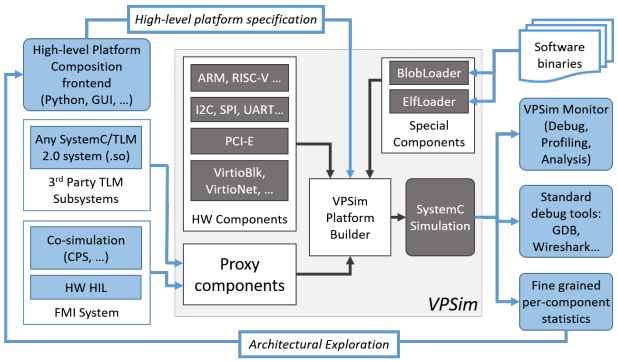


Figure 1: An Overview of the VPSim Platform.

as a Functional Mockup Unit (FMU), that can be loaded by an FMI simulation master. This makes it possible to evaluate the cyber part of a cyber-physical system (modelled in VPSim) along with its surrounding physical environment.

3.1 Component library

VPSim includes a large library of CPU models, buses and peripherals to chose from. Common controllers such as UART, I2C and SPI, PCI-Express from various vendors (Xilinx, Renesas, Cadence, etc.) are modelled in VPSim. Network, block devices and GPUs are also made available through Paravirtualization [21]. All the components that are instantiated in VPSim are fully standard-compliant SystemC/TLM 2.0 modules. CPU and Virtio models are made available to the SystemC world through a new approach for integrating QEMU into SystemC, which will be described in Section 4.

To further broaden its range of supported models, VPSim provides *proxy* components. These components are able to load and connect SystemC/TLM 2.0 initiator and target subsystems from any third-party EDA vendor to the system described in VPSim. The external subsystem is separately compiled in a shared library that implements simple glue functions. Through this interface, VPSim can seamlessly integrate CPU models such as the ARM Fast Models [20], OVP (Open Virtual Platforms) CPUs [17] or QBox [10].

3.2 Composition and Exploration

To enable truly rapid prototyping of complex architectures, it is not enough for simulations to be fast. The time required to fully describe an architecture, configure it, modify it is also of critical importance to designers and software developpers.

Most related tools employ a common scheme, wherein the platform's top is written in SystemC, either manually or generated using a Graphical User Interface, and then compiled. Later on, the configuration of the simulation is usually described in a higher level language such as Lua [10].

VPSim adopts a different philosophy: Compilation shall take place only once, and the whole simulation, including the platform to be simulated, the applications to be run, and the configuration, shall be described using a dynamic front-end. The benefits of this approach are manyfold. For instance, changes in the simulated architecture do not require recompilation. This is a true game changer at early design phases, where it is very common to make adjustments in the architecture itself. Also, the ability to describe the system hierarchy along with the components' configuration in the same environment removes the need to create a custom configuration file format for every new platform.

VPSim also provides the front-end with fine-grained simulation statistics, making it possible to perform both platform specification and Design Space Exploration within the same environment. VPSim communicates with the front-end using the XML markup language, as described in Section 4.3.

By default, VPSim ships with a Python-based composition and exploration front-end, detailed in Section 4.4. Note that unlike other tools that use high-level description languages, e.g. GEM5 [7], VPSim is not tighly linked to its Python interface, nor does it know about its existence. By using XML, which is a standardized data exchange format, many new interfaces can be developped for VPSim according to specific needs. In addition, it makes it much easier to couple VPSim with existing XML-capable tools.

3.3 Debug facilities

A decisive factor in the usefulness of any virtual prototyping tool is its ability to inspect and help understand the behaviour of the system under evaluation. VPSim offers debug and control capabilities out of the box. These are described in what follows.

3.3.1 Standard tool support. VPSim makes it possible to debug individual CPU cores in the system using GDB. Each CPU core in the system has a `gdb_enable` attribute which, when set to `True`, enables a GDB session to control and debug the code it executes.

3.3.2 The VPSim Monitor. The system designer interacts with VPSim through a command line interface named the "VPSim Monitor". At any point during the simulation, the system can be frozen using a key stroke, bringing up a command prompt. Through the Monitor, the user can inspect/alter the memory and register contents, but also reconfigure the entire simulation. For instance, it is possible to change the debug level of any component in the system to enable more or less debug information for the rest of the simulation. It is also possible to create watches on specific address ranges to monitor memory accesses, and possibly manually override the read/written values.

3.3.3 Dynamic Checkpointing. When inside the VPSim Monitor, it is possible to create one or several named checkpoints. Then, at anytime during the rest of the simulation, it is possible to roll back to the named checkpoint to investigate the cause of bugs. When a checkpoint is created, the entire user process is copied using the `fork()` system call. The parent process then sleeps and waits for a wakup signal. Rolling back to the checkpoint simply consists in terminating the current process after waking up the correct parent. The

list of checkpoints and their associated process IDs is transported with each rollback to make sure that all checkpoints are accessible from any process instance. This non-intrusive approach has several advantages. Unlike [14], our method does not require implementing a checkpointing method for each individual component, nor does it introduce changes to the SystemC kernel. Since it operates on the entire process, it works even with closed-source SystemC modules. Compared to other methods that use snapshots of the entire process, such as [11], our method can operate fully in-memory, without having to save images to disk. This makes rolling back to checkpoints created within one same simulation much faster than [11], where several seconds are necessary to store and load the snapshots.

3.4 Profiling and Performance Evaluation

Unless otherwise specified in global simulation parameters, VPSim attempts to run the target software as fast as possible. That is, most accesses to main memory will be emulated in a fast, untimed fashion based on the Direct Memory Interface (DMI) of TLM components. This mode of operations greatly eases and accelerates the functional validation and debugging phases. However, when optimizing the target software, it would also be helpful to get detailed information about its execution, such as cache usage and network latency.

One specificity of VPSim is its ability to dynamically define regions of interest, in both the software and the hardware. These regions of interest are simulated more precisely, i.e. bus accesses, caches, and other transactions are fully modelled. This helps provide a performance profile for a specific portion of the executed code.

The user can describe the beginning and end of the software region of interest dynamically (during execution), or statically (in global parameters). The hardware components that need to be precisely simulated can also be designated by the user. One common usage is when profiling a Linux user-space application. In such cases, the Linux boot phase is of little interest and can be quickly skipped through in untimed mode. VPSim then switches to a more accurate simulation mode once the user application is entered.

At the end of the software region of interest, statistics from all the simulated components are registered and displayed to the user (e.g. cache miss rate, network latency, executed instructions, etc.). This fine-grained control over the simulation's accuracy allows the user to make the best trade-offs between simulation speed and accuracy given their specific needs, which may evolve from one design phase to another.

4 KEY TECHNICAL CONTRIBUTIONS

In this section, we describe the key technical contributions that we made to enable all the features presented so far.

4.1 QEMU in VPSim

QEMU is one of the leading Dynamic Binary Translation (DBT) -based emulation solutions to date [4]. In addition to unmatched execution speeds, QEMU supports a wide range of virtualized and paravirtualized hardware models, including most of the existing CPU architectures, and many peripherals.

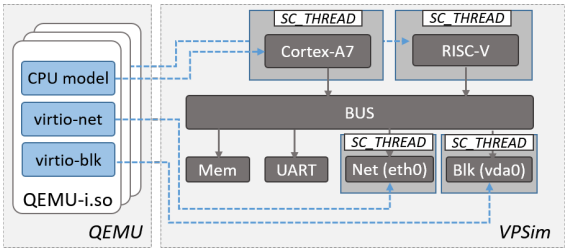


Figure 2: QEMU integration in VPSim.

Methods for leveraging QEMU in a SystemC simulation environment have been extensively explored [12], [13], [10], [15], [23], [9]. Some of these methods view QEMU as merely an Instruction Set Simulator (ISS), while others make it possible, in addition to CPU clusters, to model complete subsystems within QEMU, and some peripherals in SystemC [10]. In the latter approach, accesses to the peripherals modelled within QEMU are not visible to the outter SystemC world.

VPSim adopts a different approach. In VPSim, all CPUs and peripherals must implement a common consistent interface and be executed in the SystemC context. This is key to guaranteeing a certain level of predictability, inspectability and compatibility with VPSim's debugging and profiling tools. Therefore, in VPSim, models that are backed by QEMU can be instantiated, configured, and controlled in the same way as native components. For instance, the `VirtioNet` and `VirtioBlk` components, which are backed by virtio-net and virtio-blk devices in QEMU, can be instantiated, mapped to any address, connected to any interrupt line in VPSim. Accesses to these peripherals, as well as the interrupts they generate, are all visible and debuggable through the VPSim Monitor, for instance.

In our approach, no IO accesses are served within QEMU. Instead, all accesses are visible to the SystemC world and completed by SystemC models. Accesses to RAM (fetches and data) can be completed in QEMU for efficiency, but only if the target `Memory` module has a Direct Memory Interface (DMI) as per the SystemC/TLM 2.0 standard. QEMU's CPU execution thread and IO thread are both executed in the context of `SC_THREAD`s and `SC_METHOD`s, making simulations more predictable and controllable. In the absence of external input (Network, Keyboard, etc.), simulations in VPSim are deterministic and guaranteed to be repeatable.

While transparent to the user, internally, VPSim maintains a number of QEMU instances, as shown in Figure 2. Each QEMU instance may be used to instantiate CPUs of the same model, plus any number of peripherals. These models are then associated to proxy VPSim `components`, which are exposed to the user like any other components.

4.2 Dynamic Accuracy Control

In the SystemC/TLM 2.0 standard, some memory-mapped modules may have a DMI (Direct Memory Interface). That is, it is possible to get a pointer to their internal memory space, and access it directly, without simulating the entire TLM transaction. QEMU in VPSim makes good use of this feature by declaring all DMI regions as RAM regions [2], thereby enabling ultra-fast inline accesses to main memory.

However, to enable the accurate simulation mode presented in Section 3.4, it is necessary to perform the full TLM transactions. `Components` in VPSim possess a `DMI_OK` flag, which is active by default. When set to `false`, the component will issue a DMI invalidation request for its address range. Optionally, only a subset of the entire a accessible range can be invalidated. This propagates the invalidation request upstream and will force the upstream component to initiate full TLM transactions for the invalidated range. The upstream component then sets its own `DMI_OK` flag to false, which will provoke the invalidation of its own accessible address space by its upstream initiator, and so on until the root of the device tree is reached.

VPSim can take TLM transactions only as far as necessary to get the desired level of accuracy. For instance, if the user disables the `DMI_OK` flag only on a Cache component, VPSim only makes sure that TLM transactions get to the designated Cache component, which then completes memory accesses using DMI pointers to the final targets. VPSim's `Component` interface automatically manages these pointers internally and is able to use them to complete a transaction when eligible.

In VPSim, components take the port from which these invalidation requests are received into account. Therefore, a `Component` may force TLM accesses only on specific ports, while allowing fast DMI simulation on the rest of the device tree. This is extremely useful when only part of the memory hierarchy is of interest. For instance, forcing TLM accesses on the instruction cache `component` will only force blocking TLM transactions from QEMU when fetching instructions, while data accesses remain unaffected.

4.3 Platform Builder

At the core of the dynamic platform composition presented in Section 3.2, is a central VPSim object named the *Platform Builder*. The Platform Builder interacts with a composition front-end, such as the Python frond-end presented in Section 4.4, and has three roles:

First, it communicates to the front-end the structure of the platform description document that it expects. Currently, VPSim supports platform descriptions in XML format. Therefore, the Platform Builder automatically generates, from the list of self-registered `Components`, an XML Schema Document (XSD) that describes the structure of the expected XML document. This document also includes information about all the available components in VPSim and their attributes. The front-end should use this Schema to present the components to the user. A GUI front-end might display them as boxes that can be connected to each other, for instance. The Python frond-end presented in Section 4.4 uses the Schema to dynamically generate Python classes corresponding to each component.

Second, upon receiving an XML document from the front-end, it elaborates and configures the specified platform. Platform elaboration takes place in three phases:

- **Make**: During this phase, all the components of the platform are instantiated and initialized. The Platform Builder checks whether all the required attributes were specified, sets the specified attributes, and assigns defaults when applicable.
- **Connect**: All connections between components that were specified in the platform description are realized. The availability and compatibility of the connected ports are checked during this phase.
- **Finalize**: The Platform Builder sets some Platform-wide parameters as specified in the input XML document, and, more importantly, invokes a `finalize` callback that `Component`s may implement to perform finalization actions after all the platform has been elaborated.

Finally, at the end of a simulation, the Platform Builder collects per-component statistics and forwards them back to the front-end in an XML document. The front-end may use these statistics to build performance profiles, or for architectural exploration.

4.4 The Python front-end

The default and preferred front-end in VPSim uses the Python language. With this front-end, VPSim strives to provide a dynamic platform composition environment that is compact, intuitive, but also expressive enough to allow detailed platform configuration and description.

To illustrate with an example, Listing 1 shows how a 4-core ARMv8 Platform capable of booting the Linux operating system is described and simulated in VPSim. A simulated `system` is represented by the `System` class. Each `component` is represented by a corresponding Python class, e.g. `Memory`, `Bus`, etc. The `attributes` of each component can be either specified in the constructor for compactness (see Listing 1, Line 9), or as regular class members (see Listing 1, Line 3). This makes it possible to set or modify some attributes programmatically. Of course, each of these components has many more attributes that can be configured. When omitted, VPSim sets them to sensible defaults. Some relationships between components can also be inferred automatically. For instance, since the GIC400 (Generic Interrupt Controller) is the only interrupt controller in the system, it is automatically connected to the interrupt lines of the UART and the architected timers of the CPUs. Calling the `simulate` method interprets the entire system and launches a simulation. At the end of the simulation, per-component statistics are returned in a Python dictionary object. This model allows for as much expressiveness and modelling freedom as one would get writing a Top in SystemC. In addition, we get all the dynamicity, compactness and power of the Python language.

```
from vpsim import *

class ExampleSystem(System):
  def __init__(self):
    System.__init__(self, 'MyExample')

    self.sysbus = Bus(latency=10*ns)

    ram = Memory(base=0x10000000, size=1*GB)
    self.sysbus >> ram

    rom = Memory(base=0x00000000, size=100*KB)
    rom.is_read_only = True
    self.sysbus >> rom

    self.sysbus >> GIC400(base=0xff000000)

    self.sysbus >>
        CadenceUART(base=0xfe000000,irq=0x70)

    for i in range(4):
      cpu = Arm64(id=i,model='cortex-a57')
      cpu.reset_pc = rom.base
      cpu('icache') >> self.sysbus
      cpu('dcache') >> self.sysbus

    BlobLoader(offset=rom.base,
        file='u-boot.bin')
    BlobLoader(offset=ram.base,
        file='example.dtb')
    BlobLoader(offset=ram.base+0x80000,
        file='Image')

if __name__ == '__main__':
  sys = ExampleSystem()
  sys.addParam(param="quantum", value=1000*ns)
  stats = sys.simulate()
```

Listing 1: Simulating a quadcore ARMv8 System running Linux in VPSim

The Python front-end of VPSim was also designed with rapid design space exploration in mind. Because several System instances can live within the same Python script, and because simulation results are made available as Python objects, complete design space exploration can be performed in one same script. To accelerate DSE, the Python frond-end makes it possible to perform the simulations concurrently. To do so, it is enough to call the `simulate` method with a `wait=False` argument, to run simulations asynchronously. This fully integrated approach saves the user from having to write ad-hoc automation scripts for each new study.

5 EXPERIMENTAL SETUP AND RESULTS

In this section, we describe the experimental setup and the results obtained in terms of simulation performance.

5.1 Simulated platform and experimental setup

All the experiments have been conducted on a RISC-V-based simulated platform [6]. It is an SMP platform whose number of cores ranges from 1 to 32, running Linux kernel 4.6.2 as guest. Each simulated CPU has 32KB instruction and data caches, and all cores share an L2 cache of 2 MB and 1 GB of RAM.

This platform was simulated under VPSim on a i7-4770 host machine with 16 GB of RAM running Ubuntu 18.04 LTS. We have checked that the CPU frequency remains stable at 3.9GHz under any load.

When driven by the Accellera SystemC simulation kernel, VPSim is single-threaded and relies exclusively on single-core performance. Simulation performance is never limited by RAM, as less than 200 MB are used in any simulation scenario. The simulations were executed one at a time inside Docker containers based on Ubuntu 18.04 LTS. We have verified that Docker has no significant impact on the runtime performance of VPSim.

The simulated platform runs Linux with various benchmarks from the PARSEC [5] suite: bodytrack, ferret, swaptions, blackscholes, fluidanimate and dedup. The standard *simmedium* datasets were used. The execution sequence starts with Linux boot, followed by the benchmark setup, warmup, the benchmark's Region of Interest (ROI), teardown and finally, the platform's shutdown. Simulation time starts when the `sc_start` function is called and ends when the kernel sends the poweroff signal. Platform elaboration time is negligible with respect to the duration of a simulation.

Depending on the experiment, four accuracy modes are used:

(1) `DMI`: this is the fastest mode where as many transactions as possible are completed using DMI.
(2) `L1-ROI`: DMI is used everywhere except in the ROI for transactions initiated by CPU 0. In that case, TLM transactions reach the L1-caches, which then complete the requests using DMI.
(3) `ACCURATE-ROI`: DMI is used everywhere except in the ROI, where blocking TLM is used.
(4) `ACCURATE`: this is the most accurate mode where all memory accesses are modelled and timed using blocking TLM transactions.

`ACCURATE-ROI` mode is as accurate as `ACCURATE` mode during the ROI, which is likely to be what the user is focusing on in a real world application. `L1-ROI` mode gives accurate information about the L1-caches of a single core during the ROI. In cases where the platform and the benchmarks are symmetric in terms of executed code, the behaviour of one core with respect to caches is likely to be representative of the rest.

The number of MIPS and the real simulation time are collected at the end of each simulation. A total of 3 experiments are conducted:

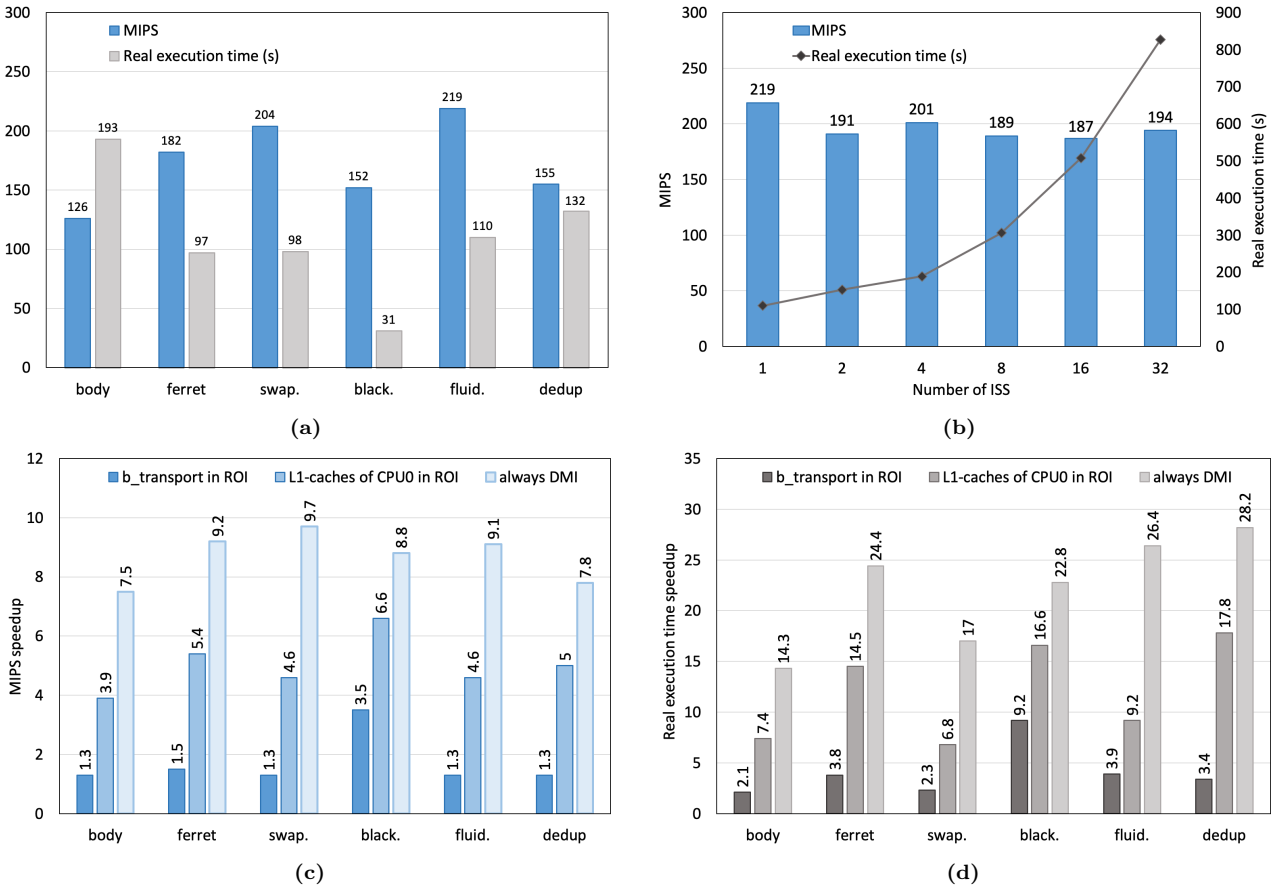


Figure 3: (a) MIPS and real simulation time for various benchmarks on a single-core platform running Linux. (b) MIPS and real simulation time for the fluidanimate benchmark with 1 to 32 cores. (c-d) Speedup with various simulation accuracy levels compared to `ACCURATE` mode on a quad-core platform.

(1) The first experiment only uses the single-core version of the platform and runs all the benchmarks to assess the performance of the simulator in `DMI` mode.
(2) The second experiment focuses on the `fluidanimate` benchmark and simulates it on platforms composed of 1, 2, 4, 8, 16 and 32 cores in `DMI` mode. It shows the effect of platform complexity on VPSim.
(3) The third experiment shows the effects of the various levels of accuracy on different PARSEC benchmarks.

5.2 Simulation results and performance

In Figure 3a, the raw performance of VPSim in fast `DMI` mode on a single-core simulated platform can be observed. The speed ranges from 121 to 219 MIPS and the real simulation time from 31 to 193 seconds. The Linux boot and shutdown processes take approximately 14 seconds. The difference in observed performance mainly comes from varying benchmark profiles and the way this impacts QEMU. Indeed, QEMU performance can be hampered by conditional jumps or memory accesses requiring the use of the software MMU. However, these factors are hard to predict and there is no clear correlation between, for instance, the frequency of load/store instructions of a benchmark and the speed at which it can be simulated.

In Figure 3b, the effect of platform complexity on MIPS and real simulation time is reported. While there is a slight drop in MIPS between the single and dual-core platforms, the speed remains in the noise margin up to 32 cores. This was expected as the overall simulation pattern remains the same: a single core doing the work outside of the ROI and all cores sharing the work inside the ROI. However, the real simulation time increases almost linearly with the number of cores. This can be explained by the sequential simulation of all cores, which introduces extra overhead in single-threaded phases (e.g. boot, setup, etc.). By contrast, the time required to simulate the ROI remains more-or-less constant, since a fixed amount of computation is shared between all cores. As a result, ROI simulation time becomes negligible compared to the other phases, hence the linear increase in simulation time.

Figures 3c and 3d show the speedups obtained when variable accuracy is applied. The reference speed for both MIPS and real simulation time is the `ACCURATE` mode. It can be observed that `ACCURATE-ROI` introduces a MIPS speedup comprised between 1.3x and 1.5x with no loss in simulation accuracy in the ROI. The only slight difference that occurs resides in the cold caches at the beginning of the ROI. However, if the ROI is big enough, this effect is negligible. Also, simulating the caches while not recording the statistics during the first steps of the ROI could mitigate this inaccuracy. The MIPS speedup obtained with `L1-ROI` ranges from 3.9x to 6.6x. In the context of cache performance evaluation on a multithreaded application, this mode may be sufficient. Indeed, if the application makes a comparable usage of all cores and exhibits negligible data sharing, then there is no need to simulate all caches.

The observation on real simulation time is comparable but speedup is larger. It ranges from 2.1x to 9.2x for `ACCURATE-ROI` and from 6.8x to 17.8x for `L1-ROI`. One particularly interesting thing to note is the difference between these speedups and the respective MIPS improvements. The reason resides in the timing accuracy variations. When simulating with maximum accuracy, all memory transactions are timed and participate in the update of the internal SystemC time, which passes faster than in `DMI` mode as a result. Consequently, all actions triggered by timed interruptions occur more often. This concerns several Linux routines such as thread scheduling, IO polling, etc. This variation in the simulated instructions is a side effect of the simulation accuracy changes. It plays in favor of variable accuracy as simulations get even shorter.

6 CONCLUSIONS

This paper introduced several contributions. First, we presented an alternative approach to integrating the QEMU emulator into SystemC. Our approach makes the CPU and peripheral models of QEMU available as TLM modules. One key benefit is the ability to leverage QEMU's high-performance emulation technologies such as Dynamic Binary Translation and Paravirtualization to boost simulation speed. Results show speeds approaching 220 MIPS when simulating workloads on top of Linux. Then, we presented a method for fine-tuning the simulation to get the best performance given specific accuracy requirements. Our framework is based on a clever use of the TLM 2.0 DMI specification. We experimentally demonstrated that by focusing simulation effort only on regions of interest, speedups of over 17x could be achieved. Finally, we showed how it was possible to abstract away all the complexity SystemC architecture modelling, by introducing a sophisticated dynamic platform elaboration infrastructure. By allowing designers to specify their modelling and exploration needs at a higher level, hours of development and compilation time can be saved. All of these promising building blocks are integrated into a unique virtual prototyping solution named VPSim.

REFERENCES

[1] TLM 2.0. 2018. Open SystemC Initiative (OSCI). https://www.accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf

[2] QEMU Memory API. 2018. QEMU. https://github.com/qemu/qemu/blob/master/docs/devel/memory.txt.

[3] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani. 2007. UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. *IEEE Computer Architecture Letters* (2007), 45–48.

[4] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference (ATEC)*. Anaheim, CA, 41–41.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. *The PARSEC Benchmark Suite: Characterization and Architectural Implications*. Technical Report TR-811-08. Princeton University.

[6] Andrew Waterman et al. 2014. The RISC-V Instruction Set Manual.

[7] Binkert et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[8] Blochwitz et al. 2012. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *International MODELICA Conference (MODELICA)*. Munich, DE, 173–184.

[9] Chiang et al. 2011. A QEMU and SystemC-based cycle-accurate ISS for performance estimation on SoC development. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 4 (2011), 593–606.

[10] Guillaume Delbergue et al. 2016. QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0. In *European Congress on Embedded Real Time Software and Systems (ERTS)*. Toulouse, FR, 315–324.

[11] Kraemer et al. 2009. A checkpoint/restore framework for SystemC-based virtual platforms. In *International Symposium on System-on-Chip (SOC)*. IEEE, Tampere, FI, 161–167.

[12] Monton et al. 2007. Mixed sw/systemc soc emulation framework. In *IEEE International Symposium on Industrial Electronics (ISIE)*. Vigo, ES, 2338–2341.

[13] Montón et al. 2009. Mixed simulation kernels for high performance virtual platforms. In *Forum on Specification & Design Languages (FDL)*. Munich, DE, 1–6.

[14] Màrius Montón et al. 2013. Checkpointing for virtual platforms and SystemC-TLM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21, 1 (2013), 133–141.

[15] M Gligor, N Fournel, and F Pétrot. 2009. Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation. In *International Conference on Hardware-Software Codesign and System Synthesis (CODES+ ISSS)*. Grenoble, FR, 71–80.

[16] C. Menard, J. Castrillon, M. Jung, and N. Wehn. 2017. System Simulation with gem5 and SystemC. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Samos, GR, 62–69.

[17] Open Virtual Platforms (OVP). 2018. Imperas Ltd. http://www.ovpworld.org

[18] Virtual System Platform. 2018. Cadence. https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/Archive/virtual_system_platform_ds.pdf

[19] Vista Virtual Prototyping. 2018. Mentor, A Siemens Business. https://www.mentor.com/esl/vista/virtual-prototyping/

[20] N Rodman. 2008. ARM fast models-virtual platforms for embedded software development. *Information Quarterly Magazine* 7, 4 (2008), 33–36.

[21] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 95–103.

[22] Simics. 2018. Wind River. http://www.windriver.com/products/simics

[23] TLMu. 2018. Edgar E. Iglesias. https://edgarigl.github.io/tlmu/tlmu.pdf

[24] N. Ventroux, A. Guerre, T. Sassolas, L. Moutaoukil, G. Blanc, C. Bechara, and R. David. 2010. SESAM: An MPSoC Simulation Environment for Dynamic Application Processing. In *IEEE International Conference on Computer and Information Technology (CIT)*. Bradford, UK, 1880–1886.

[25] Virtualizer. 2018. Synopsys. https://www.synopsys.com/verification/virtual-prototyping/virtualizer.html

[26] VLAB. 2018. ASTC. http://vlabworks.com/index.php/products/

Modeling and Virtual Prototyping for Embedded Systems on Mixed-Signal Multicores

Rodrigo Cortés Porto, Daniela Genius and Ludovic Apvrille

Modeling and Virtual Prototyping for Embedded Systems on Mixed-Signal Multicores

Rodrigo Cortés Porto
Technische Universität Kaiserslautern, Germany
LIP6 - Sorbonne Université
Paris, France

Daniela Genius
LIP6 - Sorbonne Université
Paris, France

Ludovic Apvrille
LTCI - Télécom ParisTech
Université Paris Saclay, France

ABSTRACT

This paper introduces a new approach to tackle the virtual prototyping of analog and mixed-signal embedded (AMS) systems. The application and hardware components are modeled at a high level of abstraction with SysML. From these models, a low level prototype can be generated and simulated with a cycle and bit accurate precision. This prototype combines the AMS part with a multicore platform, which controls the AMS part. As explained in the paper, the synchronization between these different Models of Computation (MoC) can be validated before the generation of the virtual prototype. A case study illustrates our approach.

ACM Reference Format:
Rodrigo Cortés Porto, Daniela Genius, and Ludovic Apvrille. 2019. Modeling and Virtual Prototyping for Embedded Systems on Mixed-Signal Multicores. In *Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '19), January 21–23, 2019, Valencia, Spain.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3300189.3300193

1 INTRODUCTION

Due to the high complexity of today's embedded systems, model-driven development techniques are a usual practice for the design and development of embedded software. These techniques rely on high level models to create a software architecture, behavior and allocation, and then perform model transformations to generate software executable code.

These approaches are however generally limited to the digital parts of the system. Yet, embedded systems are often composed of digital and analog—analog/mixed signal (AMS) and radio frequency (RF)—components. Typical examples are found in domains such as IoTs, robotics, avionics, medical and automotive.

In very early design phases, fast (but less precise) allocation exploration can be used. Assumptions and results of verifications performed at a high level of abstraction need to be cross-checked once models have been refined. To support this multi-level process, heterogeneous embedded systems may require a high-level representation including models for both AMS and RF components but also very precise simulations techniques for late validations. Last but not least, the possibility to execute software on digital parts is required as soon as possible in the design process.

This paper presents the integration of analog components into a modeling and virtual prototyping framework. The synchronization between the different Models of Computations (MoC) is performed before a virtual prototype is generated from models. The cycle-bit accurate prototyping environment can be used to feed back simulation results to higher modeling levels in order to check the taken assumptions e.g. on cache miss rate or memory access latency.

Our contribution adds analog components as targets in a MPSoC built upon general purpose CPUs running a (light) operating system and the application code.

The related work in the next section demonstrates the lack of an integrated tool offering both mixed-signal system modeling and precise simulation capabilities, as well as the possibility to run application code.

The following section presents the foundations of the present work: SystemC AMS extensions —in particular its Timed Data Flow (TDF) model of computation— and a high-level modeling and virtual prototyping tool named TTool. Our mdoeling extensions aare then presented using relevant systems with an important proportion of analog components.

2 RELATED WORK

Well established tools like *Ptolemy II* [19][22] target data-flow models for heterogeneous systems by defining several sub domains [13]. Although hierarchy is provided, instantiation of elements controlling the time synchronization between domains is left to the responsibility of designers.

Metropolis [8] is based on high level models with a clear separation between computation and communication concerns. Heterogeneous systems are taken into consideration, but heterogeneity can only be represented using processes, mediums, quantities and constraints. Hierarchical models are not allowed: all processes should be implemented in the same hierarchical level.

Metro II [11] introduces hierarchy and allows *Adaptors* for data synchronization as a bridge between the semantics of components belonging to different MoCs. The model designer must cope with the implementation of time synchronization by means of constraints, assertions, annotators and schedulers. As a common simulation kernel handles all process execution, MoCs are not well separated.

There are other frameworks based on SystemC such as HetSC [17], HetMoC [28] and ForSyDe [21], all having the disadvantage that designers must handle the instantiation of elements and synchronization aspects.

RAPIDO '19, January 21–23, 2019, Valencia, Spain

ACM ISBN 978-1-4503-6260-3/19/01…$15.00
https://doi.org/10.1145/3300189.3300193

In the scope of [1], a mixed analog-digital systems proof-of-concept simulator has been developed [12], based on the SystemC AMS extension standard [4, 18]. Another simulator is proposed in [3]. Integration with software code for general-purpose CPUs and with an operating system is however not yet addressed in these approaches.

Outside the analog/mixed signal domain, UML/SysML based modeling techniques [14, 26] are popular with industry targeting embedded systems, but are still rarely used in the domain of heterogeneous system design. Furthermore, with few exceptions such as [16, 24], they do not lower the level of abstraction to cycle bit accurate level.

3 CONTEXT

Our work is based on two foundations: the AMS extensions for SystemC and a high-level modeling and prototyping tool (TTool)

3.1 SystemC Extensions for AMS

"SystemC AMS extensions" is a standard describing an extension of SystemC with AMS and RF features [25][18]. The usual approach for modeling the digital part of heterogeneous systems with SystemC [2] is to rely on its *Discrete Event* (DE) simulation kernel. The *Timed data Flow* (TDF) model of computation (MoC) of SystemC AMS adds support for signals where data values are sampled with a constant time step. The *Electrical Linear Networks* (ELN) MoC of SystemC AMS on the other hand relies on a continuous time domain.

A TDF module is described with an attribute representing the time step and a processing function. The time step is associated to a time period during which the processing function should be executed. The *processing()* function corresponds to a mathematical function which depends on the module inputs and/or internal states. At each time step, a TDF module first reads a fixed number of samples from each of its input ports, then executes the processing function, and writes a fixed number of samples to each of its output ports. TDF modules can interact with the DE world (such as digital MPSoC platforms) using converter ports.

Figure 1 shows a TDF cluster, where the DE modules are represented as white blocks, TDF modules as gray blocks, TDF normal ports as black squares, TDF converter ports as black and white squares, DE ports as white squares and TDF signals as arrows. The TDF modules of a cluster have the following attributes:

(1) Module Timestep (**Tm**) denotes the period during which the module will be activated. One module will be activated only if there are enough samples available at its input ports.
(2) Rate (**R**). Each module will read or write a fixed number of data samples each time it is activated. This number is annotated to the ports and it is known as the *port rate*.
(3) Port Timestep (**Tp**) denotes the period during which each port of a module will be activated. It also denotes the time interval between two samples that are being read or written.
(4) Delay (**D**). A delay can be assigned to a port. As its name suggests, this attribute will make the port to handle a fixed number of samples each time it is activated, and read or write them in the following activation of the port.

Despite of all these features, [5] explains that it is hard to build a modeling environment synchronizing DE and TDF. Indeed, the TDF

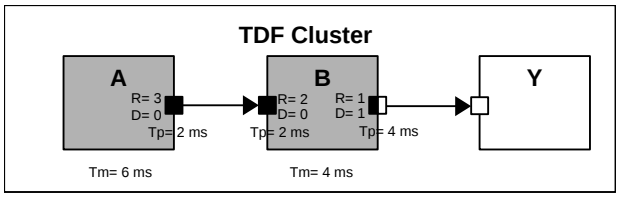


Figure 1: TDF Cluster

model of computation is based on the Synchronous Data Flow (SDF) formalism that considers models as a network of synchronous data flow blocks, and does not easily match the one of DE systems. Thus, when there are interactions between the TDF and DE models of computation, time synchronization may induce causality problems. A TDF module is connected to a DE module through converter ports. When the TDF module accesses its input converter port, the DE simulation time advances until it is equal to the TDF simulation time of the input converter port. If later an access to an output converter port occurs whose TDF simulation time is less than the new DE simulation time, a time synchronization issue will occur: the TDF simulation time of the output converter ports needs to be always greater or equal than the DE simulation time.

In the work mentioned above, this synchronization is modeled with the help of colored timed Petri Nets derived from the SystemC AMS code. Causality issues between TDF and DE MoC are then automatically checked.

3.2 TTool

TTool [7] is a SysML based, free and open-source software initially conceived for model-based engineering of (digital) embedded systems at different abstraction levels: functional, partitioning, software design and deployment. The method associated to these levels [16] details how to take hardware/software partitioning decisions at a high level of abstraction and to regularly (re)validate these decisions during software development. Software tasks for the partitioning model are captured within the functional abstraction level, and software tasks used in deployments are captured in the software design abstraction level. In both partitioning and deployment models, the computation part of tasks is allocated to processors (which can be hardware accelerators in the partitioning level), and the communication and storage part is allocated to buses and memories. An important advantage of TTool is that it offers an automated approach for formal verification and fast simulation for the digital part. Formal verification is based on internal model-checkers, or on external tools like UPPAAL [9].

From deployment diagrams, TTool can generate a virtual prototype that can be simulated with a cycle bit accurate simulator for Multi Processor Systems on Chip (MPSoC) [15]. Processor models stem from the *SoCLib* [23] public domain library written in SystemC. SoCLib targets shared-memory *multiprocessor-on-chip system* architectures based on the *Virtual Component Interconnect* [27] standard which separates the components' functionality from their communication. Software code is cross-compiled for a general purpose processor and runs under a micro kernel on the digital MPSoC.

However, while sensors, GPS, radar, etc. can be approximately modeled with highly abstracted digital blocks, the accuracy and verification would benefit from more realistic models taking into account the AMS part.

4 INTEGRATION OF ANALOG COMPONENTS INTO TTOOL

For the analog part, we focus on the Timed Data Flow (TDF) Model of Computation (MoC) which is based on the timeless Synchronous Data Flow (SDF) [20]. So-called converter ports serve as interface between the TDF and DE MoC, raising potential causality issues.

4.1 Representing Analog Components

The graphical interface of TTool has been extended by SystemC AMS DE and TDF blocks. Analog and digital parts are designed on different views. Also, there is one view per TDF cluster.

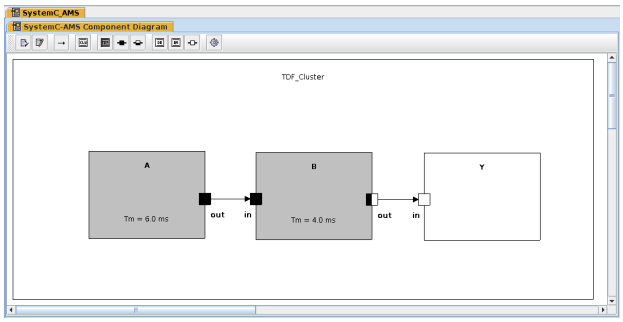


Figure 2: Extension of TTool: SystemC-AMS Diagram

Figure 2 shows a panel for the design of the introductory example. Each TDF cluster is designed in its own panel (or view) using a dedicated toolbar.

When a TDF module is created, it is possible to modify its attributes and parameters e.g. the name and Timestep or Period (Tm) of a module can be defined and time resolution selected. The parameters of a TDF module such as its internal variables or template parameters can be also set up, as shown in Figure 3.

Analog components are difficult to parameterize since they are specifically designed for one given purpose. We thus decided to provide a specific dialog window where SystemC-AMS processing() functions can be entered, e.g. see Figure 4.

From these diagrams, we can generate SystemC AMS TDF code of the components, the top cells of the mixed graphical/textual descriptions, and a Makefile.

4.2 Connecting AMS Components to the MPSoC

SoCLib is based on the shared memory paradigm, where components are interconnected based on the VCI [27] protocol. These components can be initiators which issue requests (e.g. CPUs) and targets that respond to these requests (e.g. RAM memory). The main idea for the integration of SystemC-AMS and SoCLib components into TTool is that the analog components will act as **targets**

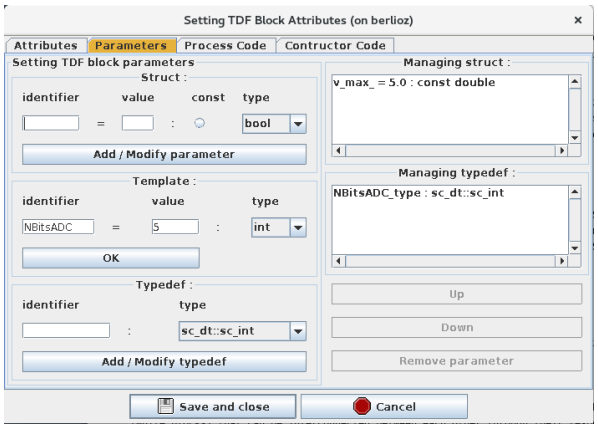


Figure 3: TDF module parameters

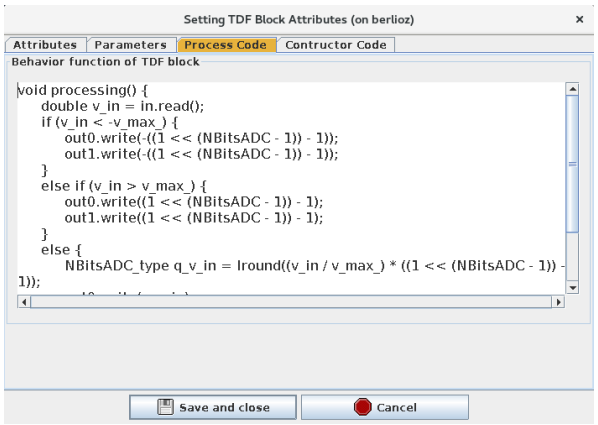


Figure 4: TDF module process code

for the SoCLib initiator digital components (CPUs, hardware accelerators, DMA, ...). The generated topcell is thus composed of SoCLib modules and interfaces to the SystemC-AMS clusters. It is also important to mention that a TDF cluster may contain DE modules which are not part of the SoCLib library.

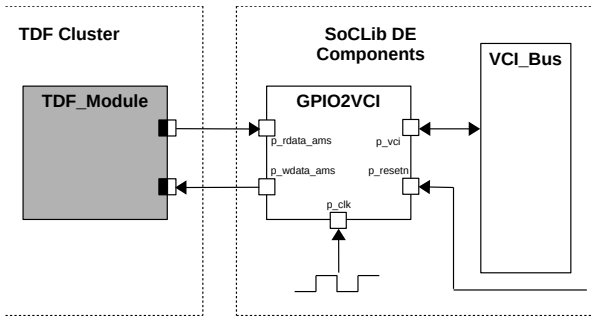


Figure 5: GPIO2VCI component

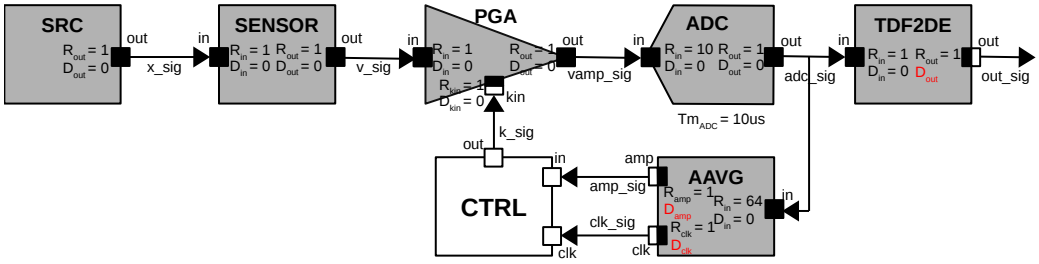


Figure 6: Vibration sensor model from [6]

Our solution is to propose a new generic adaptor module as an interface between the SystemC-AMS modules and the the SoCLib interconnect components. This adaptor is modeled as a general-purpose input/output (GPIO) adaptor to VCI. We called it **GPIO2VCI**. It fulfills the rules for writing cycle-bit precise SystemC simulation models of SoCLib. Figure 5 shows the model of this component and how it works as an interface between the SystemC-AMS modules (TDF_Module belonging to a TDF Cluster) and the SoCLib VCI interconnect component (VCI_Bus). The component is manually inserted in the graphical interface of the panel, then its instantiation and connection, in particular the required lines in the topcell, are automatically generated.

5 CASE STUDY: VIBRATION SENSOR

The model of a vibration sensor, taken from the H-Inception project [3], is shown in Figure 6. It consists of six TDF modules and one DE module.

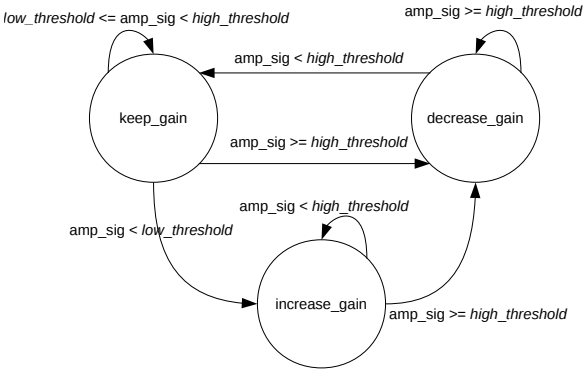


Figure 7: CTRL module state machine

The **SRC** module represents the vibration source, modeled as a generator of harmonic sinusoidal waves which represent a displacement signal (**x_sig**) caused by the vibration.

The **SENSOR** module represents a vibration sensor. It takes as input the displacement signal (**x_sig**) and yields a voltage signal (**v_sig**) which is proportional to the vibration velocity.

The programmable gain amplifier (**PGA**) amplifies the voltage input signal (**v_sig**) by a factor of 2ˆk, where k is the input value from signal **k_sig**. This signal is controlled by the gain controller DE module **CTRL**. It yields an amplified voltage signal **vamp_sig**.

The **ADC** module represents an analog to digital converter with a resolution of 5 (Nbits). The ADC has a rate of 10 in its input port. Hence, it takes 10 samples from the amplified voltage signal **vamp_sig** and produces a digitized integer value of N-bits (**adc_sig**) where the most significant bit corresponds to the sign. The Module-Timestep is assigned to this module as 10us.

The **TDF2DE** module is a converter from the TDF signal **adc_sig** to a DE signal **out_sig**. The delay **D_out**, shown in red, of its output converter port out has not been set yet.

The **AAVG** module represents an absolute amplitude averager. It calculates and outputs to the **amp_sig** the absolute average amplitudes of the received samples from the **adc_sig**. Its input port has a rate of 64, meaning that it will receive 64 samples int oder to calculate the absolute average amplitude. This module also generates a clock signal **clk_sig** at its output port clk, which has a rate of 2, meaning that a clock edge will be generated twice per activation of the module. Note that the delays **D**_clk and **D**_amp, shown in red, of its output converter ports have not been set yet.

The **CTRL** DE module represents the gain controller. This controller is modelled based on the state machine diagram shown in Figure 7. It controls the output signal **k_sig** based on the calculated absolute average amplitude given by **amp_sig**, and two given thresholds *low_threshold* and *high_threshold*.

5.1 Solving Causality Problems

The following algorithm detects causality issues and suggests how to fix them with extra delays. This algorithm is meant to be called before the generation of the virtual prototype.

```
1:  procedure DETECTTIMESYNCISSUES
2:      for each Module in Static Schedule do
3:          for each Converter Port do
4:              if Input Converter Port then
5:                  advance t_DE
6:                  compute max_t_DE
7:              else if Output Converter Port then
8:                  compute t_TDF of port
9:                  if !(t_TDF ≥ max_t_DE) then
10:                     Time synchronization issue detected
11:                     Suggest port delay to fix it
12:                 end if
13:             end if
```

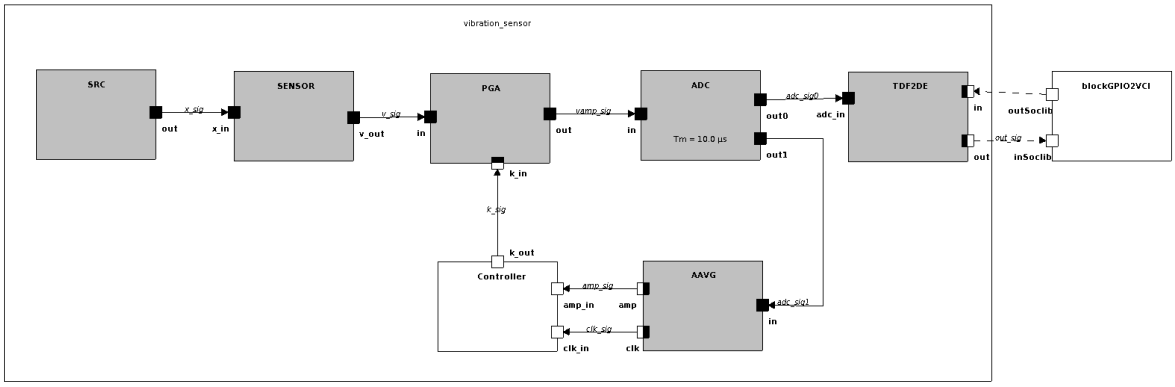


Figure 8: Vibration sensor model in TTool

14: **end for**
15: **end for**
16: **end procedure**

Based on the static schedule for one complete TDF cluster period, each time a TDF module is executed, for each accessed input converter port, the DE simulation time (t_{DE}) will advance as shown in line 5, and the maximum t_{DE} will be stored, as shown in line 6. Then, for each accessed output converter port, the TDF simulation time (t_{TDF}) is computed on line 8. The t_{TDF} of each port should be greater than or equal to the maximum stored DE simulation time, as shown in line 9. If this condition fails, it means there is a causality problem with the time synchronization and a delay in the output converter port where the issue was detected will be suggested in order to solve the problem [1].

For a first validation, the three output converter port delays shown in red (see Figure 6) were set to 0. The vibration sensor was modeled in TTool, as shown in Figure 8. Figure 9 shows the output of the validation tab of the code generation window. As time synchronization issues are found, modifications for the three delays are suggested.

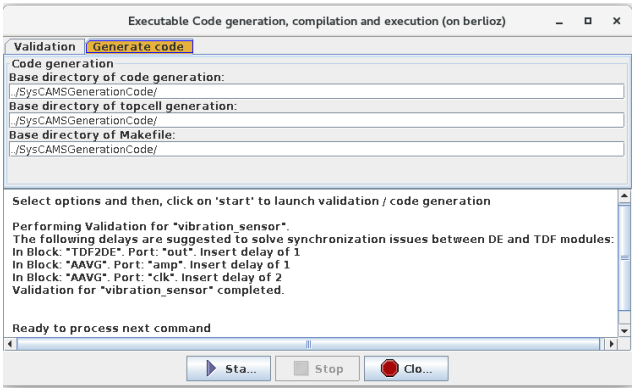


Figure 9: Code generation window with suggested delays

[1] A more detailed version of the algorithm is shown in [10].

For the output converter port out of the **TDF2DE** module, we use a delay equal to 1. For the output converter port amp of the **AAVG** module, the delay is equal to 1. Finally, 2 is used for the delay of the output converter port clk of the same **AAVG** module.

The vibration sensor model is already included in the SystemC-MDVP (Multi Domain Virtual Prototyping) simulator, developed in the context of [5], as part of the model examples. The model was simulated without giving any delays to its output converter ports. As displayed, it also suggests the three same delays as the ones suggested by TTool in order to solve the causality problems.

Finally the SystemC-AMS model taken from [3] was modified to include the same parameters as the ones used in TTool and SystemC-MDVP, i.e. the same port rates and ADC Nbits resolution. At first, the simulation was executed without assigning any delays to the output converter ports.

Synchronization issues are detected by the simulator each time the simulation runs, and delays referring to time units are suggested to solve the causality problems. First time the simulation was run, a delay of 9 μs in port tdf2de.out was suggested, corresponding to a delay of 1 since the propagated timestep of this port is 10 μs. After setting this delay, the simulation was run again; another synchronization problem was found, and a delay of 639 μs is suggested to the aavg.clk port. Since the timestep of this port is of 320 μs, a delay of 2 is needed. Finally, after setting this new delay, the simulation was run for the third time. This time, another causality problem was detected, and a delay of 639 μs is suggested to the port aavg.amp. The timestep of this port is of 640 μs, thus a delay of 1 is required.

All the delays suggested by the SystemC-AMS simulator are the same as the ones suggested by TTool and the SystemC-MDVP simulator from [5]. But on its side, TTool can identify causality problems before code generation and execution. In SystemC-MDVP, synchronization issues are found in the pre-simulation phase. That means that the SystemC-MDVP model needs to be executed only once to find any synchronization problems. In SystemC-AMS, these issues are found during the simulation phase, meaning that the simulation needs to be executed once per identified causality problem. In our case study, it needed to be executed three times.

5.2 Simulation

A simple SoC model was created as shown in Figure 10, where one SystemC-AMS Cluster block representing the vibration sensor was created, with trace file generation enabled. Such a model can potentially contain a larger number of processor cores, their number being limited only by the interconnect.

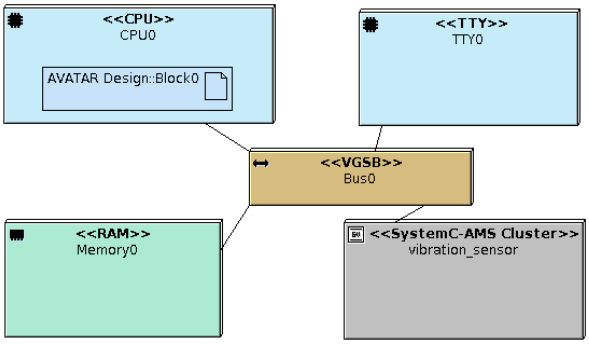


Figure 10: Deployment Diagram model including the vibration sensor TDF cluster

In order to compare the results of our approach with SystemC-AMS and SystemC-MDVP, simulations were run with the same delays and trace files with models signals were produced. Figure 11 shows the analog waveforms in blue, resulting from the simulation of the SystemC-AMS model. Figure 12 shows the analog waveforms in green, resulting from the simulation of the SystemC-MDVP model. Figure 13 shows the analog waveforms in red, resulting from the simulation of the SystemC model created from TTool. It can be seen that the outputs match, specially for the fourth signal, which corresponds to the digitized output from the **ADC** component. The first waveform corresponds to the signal **x_sig** which carries the output of the harmonic sinusoidal wavelets generator **SRC**, simulating a vibration source. The second waveform is from the signal **v_sig**, which is the voltage output from the vibration sensor module **SENSOR**. The third waveform corresponds to the **v_amp** signal, which is the signal being amplified by the **PGA** module. The fourth signal **adc_sig** is the digitized output from the **ADC** module. The fifth signal **amp_sig** corresponds to the output of the absolute amplitude averager **AAVG** module which is connected to the DE controller **CTRL**. This controller emits the sixth signal **k_sig** which carries the factor that will be used by the **PGA** module to amplify the voltage signal **v_sig**. The last signal is the **clk_sig** used as clock signal for the controller **CTRL** module. Note that the **amp_sig** and **k_sig** signals from the SystemC-MDVP simulation look different: this is due to the generated trace file that didn't create values when they were repeated. So only the value changes are shown, but they still correspond to the outputs from the other traces.

This case study demonstrates that the solution implemented in TTool to detect time synchronization issues yields the same results as the ones suggested by the SystemC-AMS and the SystemC-MDVP simulators. Moreover, the time synchronization issues detection is performed at the design level, before the virtual prototype or the software code are generated, thus giving the designer the possibility to adapt their design before simulation.

6 CONCLUSION AND PERSPECTIVES

We integrate SystemC-AMS (TDF) components into a High-level modeling tool for complex embedded systems and show how application code can be run in such systems by combining the AMS part with a prototype built out of SoCLib components. Contrary to other approaches, we detect causality issues between the two parts of the simulation before any code is generated, which is one of the major strength of our approach.

To do so, we created a library to provide read and write functions to the GPIO2VCI component, which can be used in the State Machine Diagrams of TTool. The AMS hardware components are considered to be targets inside the MPSoC platform. In the future, we suggest to authorize these components to act as initiators, or to support interrupts. Also, the static schedule computed by TTool could be optimized, so that the suggested delays to solve time synchronization issues are minimum.

The *Electrical Linear Networks* (ELN) model of computation of System-C AMS relies on a continuous time domain. We plan to push our studies further by integrating ELN into our tool.

Finally, even if analog components tend to be unique, for typical components such as filters, analog/digital converters, sine sources, we plan to provide a library of parametrizable building blocks.

REFERENCES

[1] *Beyond Dreams (Design Refinement of Embedded Analogue and Mixed-Signal Systems), http://projects.eas.iis.fraunhofer.de/beyonddreams/site/index/index_en.html.*
[2] SystemC. In *http://www.systemc.org*.
[3] *Heterogeneous Inception*, 2012-2015. https://www-soc.lip6.fr/trac/hinception.
[4] Accellera systems initiative. *SystemC AMS extensions Users Guide, Version 1.0.* Accellera systems initiative, March 2010.
[5] Andrade, L., Maehne, T., Vachoux, A., Ben Aoun, C., Pêcheux, F., and Louërat, M.-M. Pre-Simulation Formal Analysis of Synchronization Issues between Discrete Event and Timed Data Flow Models of Computation. In *Design, Automation and Test in Europe, DATE Conference* (Mar. 2015).
[6] Andrade Porras, L. *Principles and implementation of a generic synchronization interface between SystemC AMS models of computation for the virtual prototyping of multi-disciplinary systems.* PhD thesis, Université Pierre et Marie Curie, 2016.
[7] Apvrille, L. *TTool, an open-source toolkit for the modeling and verification of embedded systems.* http://ttool.telecom-paristech.fr/.
[8] Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., and Sangiovanni-Vincentelli, A. L. Metropolis: An integrated electronic system design environment. *IEEE Computer 36*, 4 (2003), 45–52.
[9] Bengtsson, J., and Yi., W. Timed automata: Semantics, Algorithms and Tools. In *Lecture Notes on Concurrency and Petri Nets* (2004), W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag, pp. 87–124.
[10] Cortés Porto, R. Integration of SystemC-AMS simulation platforms into TTool. Master's thesis, Technische Universität Kaiserslautern, november 2018.
[11] Davare, A., Densmore, D., Meyerowitz, T., Pinto, A., Sangiovanni-Vincentelli, A., Yang, G., Zeng, H., and Zhu, Q. A next-generation design framework for platform-based design. In *Conference on using hardware design and verification languages (DVCon)* (2007), vol. 152.
[12] Einwich, K. *SystemC AMS PoC2.1 Library, COSEDA, Dresden*, 2016. http://www.coseda-tech.com/systemc-ams-proof-of-concept/.
[13] Fong, C. Discrete-time dataflow models for visual simulation in Ptolemy II. *Master's Report, Memorandum UCB/ERL M 1* (2001).
[14] Gamatié, A., Beux, S. L., Piel, É., Atitallah, R. B., Etien, A., Marquet, P., and Dekeyser, J.-L. A model-driven design framework for massively parallel embedded systems. *ACM Trans. Embedded Comput. Syst 10*, 4 (2011), 39.
[15] Genius, D., and Apvrille, L. Virtual yet precise prototyping: An automotive case study. In *ERTSS'2016* (Toulouse, Jan. 2016).
[16] Genius, D., Li, L. W., and Apvrille, L. Model-Driven Performance Evaluation and Formal Verification for Multi-level Embedded System Design. In *Conf. on Model-Driven Engineering and Software Development (Modelsward)* (Porto, 2017).
[17] Herrera, F., and Villar, E. A framework for heterogeneous specification and design of electronic embedded systems in systemc. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 12*, 3 (2007), 22.
[18] IEEE. *IEEE Std 1666.1 standard*, January 2016.
[19] Lee, E. A. Disciplined heterogeneous modeling. In *Proceedings of the ACM/IEEE*

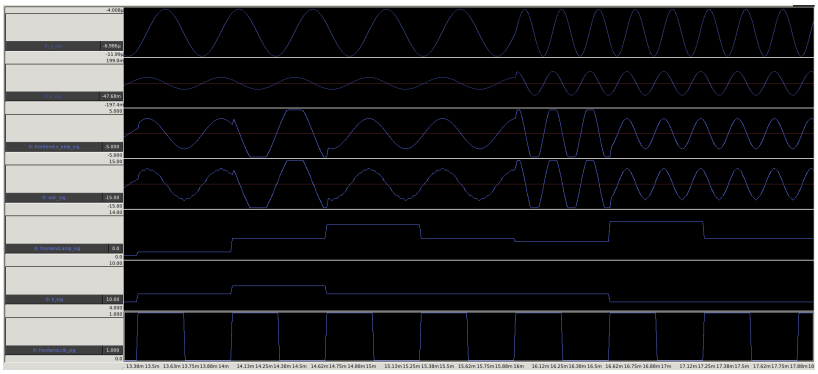


Figure 11: Vibration sensor trace signal from SystemC-AMS simulation

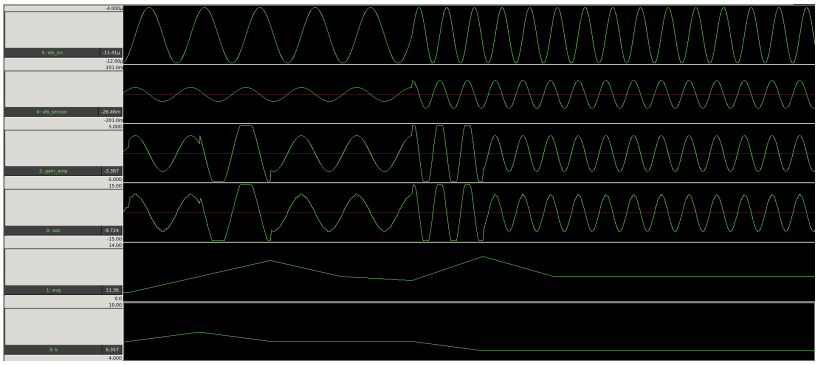


Figure 12: Vibration sensor trace signal from SystemC-MDVP simulation

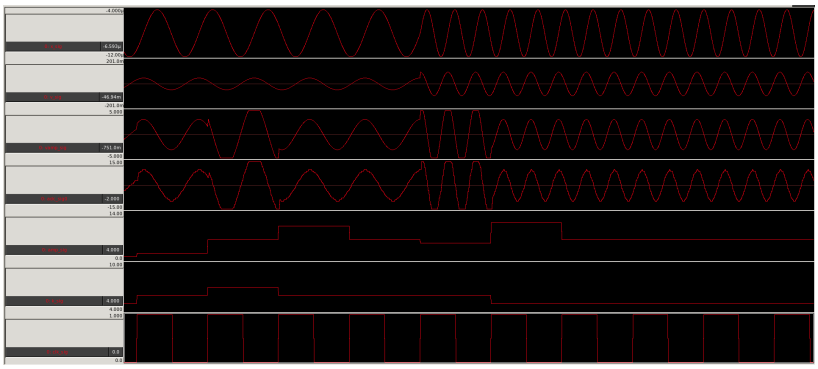


Figure 13: Vibration sensor trace signal generated from TTool's simulation

13th International Conference on Model Driven Engineering, Languages, and Systems (MODELS) (Oct. 2010), D. Petriu, N. Rouquette, and O. Haugen, Eds., LNCS 6395, Springer-Verlag, pp. 273–287.

[20] Lee, E. A., and Messerschmitt, D. G. Synchronous Data Flow. *Proceedings of the IEEE 75*, 9 (1987), 1235–1245.

[21] Niaki, S. H. A., Jakobsen, M. K., Sulonen, T., and Sander, I. Formal heterogeneous system modeling with SystemC. In *Forum on Specification and Design Languages (FDL)* (2012), IEEE, pp. 160–167.

[22] Ptolemy.org, Ed. *System Design, Modeling, and Simulation using Ptolemy II*. 2014. http://ptolemy.org/books/Systems.

[23] SoCLib consortium. The SoCLib project: An integrated system-on-chip modelling and simulation platform. Tech. rep., CNRS, 2003. www.soclib.fr.

[24] Taha, S., Radermacher, A., and Gérard, S. An entirely model-based framework for hardware design and simulation. In *DIPES/BICC* (2010), vol. 329 of *IFIP Advances in Information and Communication Technology*, Springer, pp. 31–42.

[25] Vachoux, A., Grimm, C., and Einwich, K. Analog and mixed signal modelling with SystemC-AMS. In *ISCAS (3)* (2003), IEEE, pp. 914–917.

[26] Vidal, J., de Lamotte, F., Gogniat, G., Soulard, P., and Diguet, J.-P. A co-design approach for embedded system modeling and code generation with UML and MARTE. In *DATE* (2009), IEEE, pp. 226–231.

[27] VSI Alliance. Virtual Component Interface Standard (OCB 2 2.0). Tech. rep., VSI Alliance, Aug. 2000.

[28] Zhu, J., Sander, I., and Jantsch, A. Hetmoc: Heterogeneous modelling in SystemC. In *Forum on Specification & Design Languages* (2010), IET, pp. 1–6.

Performance Modeling of MPI-based Applications on Cloud Multicore Servers

Abdallah Saad, Ahmed El-Mahdy and Hisham El-Shishiny

Performance Modeling of MPI-based Applications on Cloud Multicore Servers*

Abdallah Saad†
Egypt-Japan University of Science and Technology
New Burj El-Arab, Alexandria
abdallah.saad@ejust.edu.eg

Ahmed El-Mahdy‡
Egypt-Japan University of Science and Technology
New Burj El-Arab, Alexandria, Egypt
ahmed.elmahdy@ejust.edu.eg

Hisham El-Shishiny§
shishiny@eg.ibm.com

ABSTRACT

While cloud computing is widely adopted in many application domains, it is not yet the case for the high performance computing (HPC) domain. HPC traditionally runs on homogeneous, high-cost servers with fast networking providing for predictable performance; while bare-metal cloud offerings is promising, the underlying hardware is heterogeneous, with slower network connection, making it difficult to predict performance and hence tune applications. In this paper we consider performance modelling message passing interface (MPI)-based applications, being a major class of HPC applications. In particular, we present a queueing network performance model to account for computation and communication contentions on the underlying heterogeneous, relatively slow-interconnect architecture of the cloud bare-metal servers. The proposed model uses a non-linear problem solver to enhance the parameters acquired by profiling. We utilise our model to conduct an initial study of the performance of two benchmarks from SPECMPI-2007 suite and two NASA Parallel kernels, executing on a small cluster with varying number of multicore servers ranging from 2 to 8. Comparing the predicted and actual execution times of workloads with different number of processes shows 86% average accuracy for the benchmarks used.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**; • **Networks → Network performance modeling**; *Network performance analysis*; • **Computing methodologies → Modeling methodologies**; • **Mathematics of computing →** *Nonlinear equations*;

KEYWORDS

Performance Modelling, Message Passing Applications, Cloud Computing, High Performance Computing, Bare Metal Service

ACM Reference Format:
Abdallah Saad, Ahmed El-Mahdy, and Hisham El-Shishiny. 2019. Performance Modeling of MPI-based Applications on Cloud Multicore Servers. In *Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '19), January 21–23, 2019, Valencia, Spain.* ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3300189.3300194

1 INTRODUCTION

Cloud computing has demonstrated its efficiency and benefit for several applications due to its elasticity, easy of usage, scalability, availability and its 'pay as you go' business model. Also, the cloud is able to produce a better turnaround time than the on-premise HPC clusters, considering the waiting time by cluster management systems [8].

However, its utility for HPC applications is still questionable as the HPC users face two main challenges [9]: the relatively higher network latency than the on-premise clusters' network that limits the scalability of the tightly coupled parallel applications, and the difficulty to estimate the cost of running HPC applications on the cloud unpredictable environment. Egwutuoha et al. [4] recommended to use bare-metal servers instead of the cloud instances when running the tightly coupled parallel applications. Currently many cloud providers which include IBM, Amazon, Alibaba, offer dedicated bare-metal cloud machines for HPC applications. However, bare-metal servers still suffer from being heterogeneous, and of relatively lower networking speeds [10] that increases the contention on the network queues. Such characteristics make performance and cost prediction more complicated, making it more difficult to performance/cost optimise parallel applications for such cloud environment.

In this paper we propose an analytical performance model based on queueing networks to capture the underlying resource contention for the underlying heterogeneous architecture. The model targets the important class of MPI applications, which is typical for the HPC domain. The proposed performance model is used to predict the execution time of parallel MPI-based applications running on multicore servers. The model parameters are acquired using a non-linear solver, allowing for general applicability. Then, the prediction procedure can be done in a very short time for different configurations of parameters allowing for a better exploration for the design space. The model helps in answering several questions such as what is the turning point where increasing the scalability of

*Produces the permission block, and copyright information
†On leaving from Benha University, abdallah.mohamed@feng.bu.edu.eg.
‡On leaving from Alexandria University.
§Former Manager of Advanced Technology and IBM Center for Advanced Studies in Cairo.

RAPIDO '19, January 21–23, 2019, Valencia, Spain

ACM ISBN 978-1-4503-6260-3/19/01…$15.00
https://doi.org/10.1145/3300189.3300194

the parallel program is worthless from the performance/cost view point?; what is the best grouping of servers that fits the workload needs given the communication distances between the servers and the processing powers of each?; and how the distribution of processes on servers affects performance, especially that the number of cores per server can vary?

To verify the proposed model, an initial experiment is conducted on three different clusters sizes of cloud physical machines of two, four and eight nodes. Two benchmarks of the SPECMPI-2007 benchmark suite, and two NASA parallel kernels are used as the HPC MPI-based workloads. Results show an average accuracy of 86% when comparing predicted with actual execution times on different configurations. Also, this work is compared with a closely related work [14] using two NASA parallel benchmarks [3] running on the different size clusters. For configurations with homogeneous network topology (single process per node) both models achieve similar accuracy; however, for heterogeneous topology (multiple processes per node) the accuracy of the other model degrades significantly.

The rest of this paper is organised as follows: related work is presented in Section 2. Section 3 describes the proposed performance model and its corresponding parameters acquisition. The methodology used in the modelling process is depicted in Section 4. Section 5 presents and discusses the model validation experiments. Finally, Section 6 concludes the paper and discusses future work.

2 RELATED WORK

Several work is done to evaluate the usage of the cloud as an alternative to ordinary HPC systems. Jackson et al. [7, 11] and Zhai et al. [18] examined the usefulness of cloud computing for e-Science and HPC applications, and Wang et al. [16] studied the impact of virtualisation on network performance of Amazon EC2 data centre. Gupta et al. [6] evaluated the performance of HPC applications on different execution platforms; cluster, grid and a private cloud. Their work shows the performance bottleneck caused by communication on the cloud even with dedicated 10 Gigabit Ethernet network medium. Also, they demonstrate how the performance instability and unpredictability affect the cost of long running applications. Marathe et al. [8] investigate the cloud and HPC clusters using different metrics of comparison; turnaround time and cost. their results show the inefficiency of communication intensive applications on Amazon EC2.

Another trend is to model the cloud system for performance prediction of running HPC applications. Shi et al. [14] introduce an instrumentation assisted complexity analysis methodology for program scalability analysis. Their methodology is based on Amdahl's and Gustafson's laws. They manage to predict the execution time for parallel programs on cloud and HPC cluster. Their results show 71% average accuracy between actual and predicted execution times of five NAS benchmark 'kernels' on HPC cluster, where our proposed model achieves 82.5% average accuracy of predicting execution of two SPECMPI benchmarks. Our model prediction seems better on the overall results. However, the clusters size used in experimentation of the proposed model is smaller than the size of the used cluster in their experiments.

Beyond the cloud virtualised environment, Egwutuoha et al. [4] recommends to run the tightly coupled parallel applications on a bare-metal cluster of servers on the cloud to gain the benefit of the cloud availability—small queue waiting time to start the requested service—and to avoid the cloud relatively slow network compared to HPC clusters.

In order to predict the scalability of parallel programs, Barnes et al. [1] use a modified regression model. In their work, they use several program executions on different number of working processing nodes as a training test to extrapolate the performance of running the parallel program with untrained configuration of processing nodes' count. Their model shows accurate best-fit predictions. However, their work is not flexible to explore and explain the effect of changes in the parameters of interest, e.g. number of cores used per server, allocation of servers in the underlying network, the current network state and the amount of currently available processing power per server. Also, Bridges et al. [2] provide a work in progress to model the MPI main performance characteristics of communication operations using a simple closed queuing network model. They validate their work using simple communication benchmarks running on a tiny cluster of two multicore machines. Their work focuses on modelling the communication operations and does not include modelling neither the hardware characteristics, nor the computation operations of the parallel programs.

3 PERFORMANCE MODEL OVERVIEW

The heterogeneity of the underlying hardware and the underlying relatively slow network would potentially result in contention on both the CPUs and network; the model therefore considers queueing networks for modelling such contentions. Moreover, to generalise the applicability of our model, we decompose/separate the model into two main components: one for modelling the running workload (software), and the other for modelling the underlying system (hardware). This provides for larger design space exploration.

This section is organized as follows: Section 3.1 discusses the workload model; Section 3.2 describes the underlying system and its interaction with the workload model; Sections 3.3, 3.4 and 3.5 discuss model parameters acquisition.

3.1 Workload modelling

We consider the workload application as a long sequence of instructions, w, running on n processes. Each process is repeatedly invoked to perform a number of instructions ϕ followed by a send and receive operation. Every single invocation of the process is called a job[1]. The number of jobs invoked during the program running time equals the number of sends per process s. Accordingly, we have the following equation holding:

$$w = \phi n s(n) \tag{1}$$

Where s and w are functions of n, depending on the current benchmark.

We consider the MPI program to have a fixed number of cycles;

[1]This is not to be confused with an MPI job; a job here refers to a queueing network job, which is a process cycle

each cycle is a sequence of computations ϕ that ends with a communication operation. When n processes are running, the cycles are distributed uniformly over the processes as jobs.

3.2 Queueing Network and process life cycle

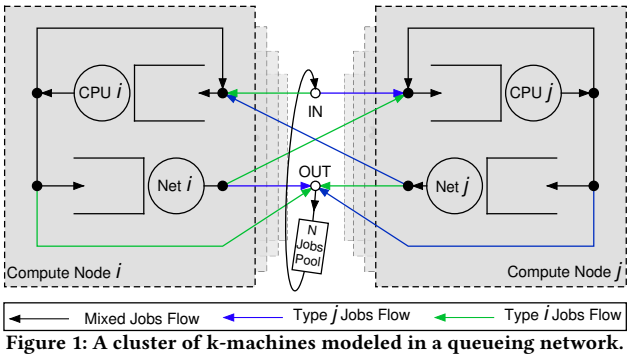


Figure 1: A cluster of k-machines modeled in a queueing network.

Fig. 1 presents our closed-queueing network system for modelling the execution of an MPI program on a cluster of k multicore machines. The figure shows the queueing network for two arbitrary nodes i and j. All other nodes on the cluster have the same structure; all nodes share a centralised job pool. A job represents a single invocation of a process (a cycle). Initially, jobs reside in an MPI jobs pool, which is a collection of jobs ready for execution.

A job enters the system through the 'IN' port, and eventually exits through the 'OUT' port and returns back to the pool, completing an execution cycle. Then, a job is scheduled immediately for another cycle, and this process repeats.

During the job life cycle, the job is distributed over one of the k processing nodes (node i). The job starts using the processing powers of node i through CPU i queue to perform its ϕ computation sequence. After that, the job begins a communication operation with another corresponding job located either locally on node i or remotely on a different processing node (node j). The local communication operations are performed on the CPU queue as a communication overhead quanta, whereas the remote communication operations are done through the processing node's network queues of the two communicating nodes, Net i and Net j.

3.3 Service time calculation

The service time is the average time required to serve a job visiting a queue. Thus, for the case of the CPU queues, the service time is calculated such that it is the time required to process the ϕ instructions of a job. Thus, the service time of the CPU can be computed as: $s_{\text{cpu}} = k\phi$, Where k is a given constant representing the time to execute a single instruction.

Now, assuming that the CPU has c cores, the service time will improve with increasing number of jobs inside the CPU queue, n, until reaching the server's capacity. Thus:

$$s_{\text{cpu}} = \frac{k\phi}{\min(n, c)} \tag{2}$$

Assuming a fixed workload and substituting for $\phi = w/ns(n)$, we get:

$$s_{\text{cpu}} = \frac{kw}{s(n)n\min(n, c)} \tag{3}$$

Now, consider the network part where the service time of the Net queue is the time required to communicate an average size message between two nodes. We define the average message size, m, as a function of n; and the number of total messages is given by $s(n)n$. Therefore, the mean service time of the network can be represented as the communication time of the average message size for a given number of processes. The simplified cost model for communicating messages [5] is used to compute the communication time as follows:

$$s_{\text{net}} = T_s + m(n)T_w \tag{4}$$

Where, T_s is the start-up time to handle a message at the two communicating nodes, and T_w is the per word transfer time. To simplify the model, we assume $T_s = 0$, and $m(n) = m_a/n + m_b$. Thus,

$$s_{\text{net}} = (m_a/n + m_b)T_w \tag{5}$$

m_a and m_b are parameters of modelling the message size variation with respect to n. These parameters can be calculated by fitting the workload profiling data that contains the workload's communication behaviour with respect to n. Where, T_w can be obtained using the methodology described in [13]. In order to calculate the constants values in the CPU service time equation, k and w (for a fixed size workload), we give these constants initial values and consider the outcome service time as an initial guess to a non-linear problem solver, e.g. Gauss-Newton Algorithm [17], to get the best fit for these constants. Thus, $s_{\text{CPU}_{\text{guess}}}$ and $s_{\text{Net}_{\text{guess}}}$ can be defined respectively as;

$$s_{\text{CPU\_guess}} = \frac{\text{CPU\_Constant}}{s(n)n\min(n, c)} \tag{6}$$

$$s_{\text{Net\_guess}} = s_{\text{net}} \times \text{Net\_Constant} \tag{7}$$

Where, CPU_Constant = kw and Net_Constant represents the imperfection in probing the network state.

3.4 Visit ratio calculations

The visit ratio of a queue is the average number of times a job will visit the queue during its life cycle. In other words, it is the number of quanta of processing/communication required by a job assuming that each visit takes a certain time quantum.

In order to calculate a visit ratio, we need to consider the job's average CPU quanta spent during both its computation and the communication overhead processing. We start by calculating the communication overhead by profiling the parallel application on a physical machine under no contention condition; where there is no network communication time and all jobs are communicating locally.

Now, let us consider the time required to finish a job life cycle (Cycle_time); it consists of the time to execute the job's instructions (T_{comp}), the network time (T_{nw}) the job needs to communicate remotely, and the communication overhead time (T_{oh}) that mimics the communication waiting time the job spends waiting for its adjacent job to prepare the response of the communicated message. Thus, we can define: Cycle_time = $T_{\text{comp}} + T_{\text{nw}} + T_{\text{oh}}$. And thus, the cycle time for each process (Process_cycle_time) is defined as: Process_cycle_time = Cycle_time/n.

By profiling the parallel application using a number of running processes that is less than or equal to the available processing units (no CPU contention) we get the execution time of the running application (T_{exe}), and the wait time for all the MPI communication operations (MPI_Wait) performed by all processes. For that parallel execution, we assume all processes to start and end execution together. According to that, the application execution time represents the running time of a single process. Thus, the per process cycle time is defined as Process_cycle_time = T_{exe}/s.

Since the profiled run was on a single machine, the T_{nw} is neglected. And T_{oh} from profiling can be represented as; T_{oh} = MPI_Wait/s. Thus, in this case T_{comp} is defined as $T_{\text{comp}} = T_{\text{exe}} - T_{\text{oh}}$.

By knowing T_{exe}, T_{comp} and T_{oh}, we can compute the sub-visit ratio of computation time spent during the job cycle time to the total cycle execution time (V_{comp}). And similarly, the sub-visit ratio of communication overhead time of a job cycle to the cycle's total execution time (V_{comm}). Where, $V_{\text{comp}} = T_{\text{comp}}/T_{\text{exe}}$ and $V_{\text{comm}} = T_{\text{oh}}/T_{\text{exe}}$. After calculating the sub-visit ratios needed, V_{comp} and V_{comm}, we can deduce the job visit ratio to each processing node's CPU queue. Let's assume a compute node i; the CPU queue of node i is visited by all the jobs located originally on node i to perform their computation. Also, the jobs located on node i would re-enter the CPU queue again for doing the communication overhead if these jobs are going to communicate locally. Finally, the CPU queue of node i is possibly visited by jobs located originally on different compute nodes to make remote communication with adjacent jobs on node i. Thus, the visit ratio of the CPU queue of node i is defined as:

$$V_{\text{CPU}_i} = \frac{n_i}{n} V_{\text{comp}} + \frac{n_i}{n} \frac{n_i - 1}{n} V_{\text{comm}} + \frac{n - n_i}{n} \frac{n_i}{n} V_{\text{comm}} \quad (8)$$

Where $\frac{n_i}{n}$ is the ratio of node i jobs to the total number of running jobs, $\frac{n_i}{n} \frac{n_i - 1}{n}$ is the ratio of jobs of node i that possibly could communicate locally with jobs of node i as well, and $\frac{n-n_i}{n} \frac{n_i}{n}$ is the ratio of jobs originally located outside node i and performing remote communication operations with jobs of node i.

Similarly, the visit ratio for the network device queue (V_{net_i}) can be deduced. The jobs of node i that are communicating remotely visit node i's network queue (net_i). Also, jobs located outside node i but communicating remotely with jobs of node i are going to visit net_i in their return journey. Thus, we can define V_{net_i} to be:

$$V_{\text{net}_i} = \frac{n_i}{n} \frac{n - n_i}{n} + \frac{n - n_i}{n} \frac{n_i}{n} \quad (9)$$

Where $\frac{n_i}{n} \frac{n-n_i}{n}$ represents the ratio of jobs of node i doing remote communication with jobs of other nodes, and $\frac{n-n_i}{n} \frac{n_i}{n}$ represents the ratio of jobs of nodes other than i that are communicating remotely with jobs of node i.

3.5 Job response time calculations

Now let us call the system response time to be R; it represents the execution time for a ϕ sequence. Since every process has $s(n)$ sequence of ϕ to execute, each process response time is $s(n)\phi$. And since we assume that all n parallel sequences enter the system at the same time and finish execution at the same time, the system total execution time, T, for the w instructions is:

$$T = Rs(n) \quad (10)$$

We profile the workload for different number of uniformly distributed running processes, to get the execution time for each and to profile the communication operations to help modelling them for any given number of running processes. For example, modelling the average size of messages for different n, $m(n)$ as An^{-B} and the average number of sends per process $s(n)$ as $C \ln(n) + D$. Where $A, B, C$and D are constants calculated using the gathered profiling data. Using these profiling data, the CPU_Constant and Net_Constant are calculated using GNA nonlinear problem solver.

Using the service time and visit ratio, the average response time of jobs (R) to complete one life cycle is calculated. In our model, we use a numerical solution to get R while the workload is parallelised over n processes. Mean value analysis (MVA) algorithm [12] is used to calculate R, knowing the values of service time and visit ratio for all queues in our queueing network model.

4 PROPOSED SYSTEM DESIGN

The proposed system shown in Fig. 2 is used to model both the multicore system available resources and the MPI-based application required resources as well. The system starts with two parallel steps:

- the application profiler runs the MPI-based application (more than two times) with test input data to profile the program communication behaviour with respect to the variation in number of working processes,
- the hardware prober gathers information about the available resources and its current state.

If these information are known a priori, then those parallel steps are skipped. Then, the output from the application profiler and the hardware prober move to the initial model for calculating the guess service time and visit ratio for each CPU queue and NW queue for each compute node in the system. At that instant, the system has two paths to go based on the availability of the final CPU and NW constants:

- if available; the mathematical model uses them to calculate the service time and visit ratio for each queue in the queuing network. Afterwards, the mean value analysis (MVA) technique is used to numerically estimate the response time of the system for any given number of processes, then, print it to the user and the system exit.
- if not available; the non-linear problem solver, Gauss-Newton Algorithm (GNA), is used to find the values of the constants that minimise the error between the expected response time and the execution times previously measured during profiling. To do that, the GNA uses the proposed model equations with the initial service times and sub-visit ratios to get the temporal service times and visit ratios for each queue in the queuing network. Then, it passes them to the MVA technique to calculate the predicted response time and send it as a feedback to the GNA. If the error is less than a predefined threshold ϵ, these constants' values (for this workload on this multicore system at this state) are stored and the proposed system finishes. Otherwise, the system repeats that path until the system finds the constants that minimise the error to be less than ϵ.

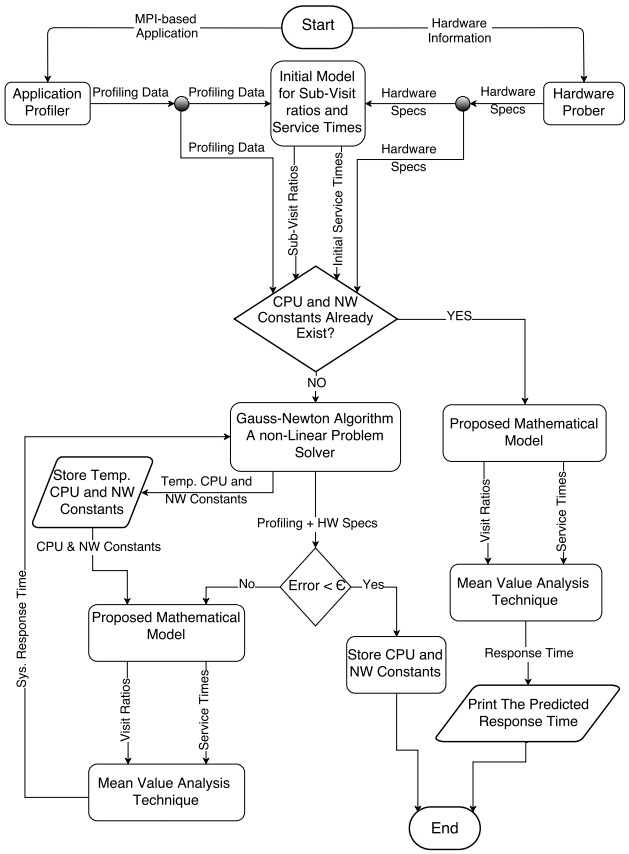


Figure 2: Proposed system design

5 EXPERIMENTS, RESULTS AND ANALYSIS

In this work, the '126.lammps' and '130.socorro' benchmarks of SPECMPI-2007 benchmark suite [15] are used as the MPI-based workload, as well as two kernels from NASA Parallel benchmark. Clusters of 2, 4 and 8 multicore compute nodes are used as the underlying system with 1 GBit/s Ethernet network. All of the compute nodes are running CentOS-7.4 release. The clusters specifications are shown in Table 1. The system runs MPICH 3.1 and gcc 4.8.5 tool chain.

Table 1: Clusters Specifications

Num of nodes	Nodes variability	Memory Size (GB)	Processors model	Num of cores
2	homogeneous	24	E5620	16
4	homogeneous	24	E5620	32
8	4x	24	E5620	88
	2x	48	E5645	
	1x	24	E5-2450	
	1x	32	E5-2450	

After gathering and calculating all the data needed to calculate the service time and the visit ratio to the queueing network model queues, both service times and visit ratios are used as an input to the MVA algorithm to calculate the response time R numerically. Finally, from Equ. 10 the predicted execution time is calculated for the number of running processes n.

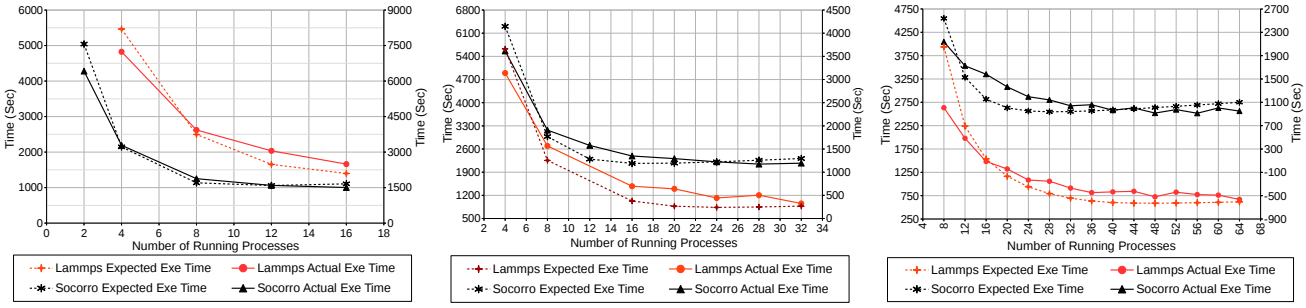
Fig. 3a, Fig. 3b and Fig. 3c show a comparison between our model prediction and the actual measurements of running 'lammps' and 'socorro' benchmarks on the cluster of two, four and eight nodes, respectively. The benchmarks are run with medium reference (mref) data set as it is more reliable workload than the test data size that has a very short execution time which is more susceptible to noise. For all sub-figures in Fig. 3, The secondary Y-axis represents the actual and predicted execution time of the 'Socorro' benchmark, where the primary Y-axis represents the execution time for 'Lammps' benchmark. The results illustrate the model prediction for different parallel workloads.

In order to measure the model accuracy, two statistics are used; coefficient of variation (CV) and mean absolute percentage error (MAPE). The Accuracy is calculated by subtracting MAPE from 100. On average for the three different clusters, the predicted execution time for 'Socorro' was accurate by 89.8% with average CV = 0.16 and, 'lammps' achieves 82.3% accuracy with average CV = 0.24. Also, Fig. 3 shows the turning points at which scaling up the parallelism of the program do not have a significant impact on the performance. For a cluster of two nodes, lammps execution time starts to saturate using 16 cores while socorro saturates using 8 cores only. For four nodes the saturation happens at 24 and 16 cores for lammps and socorro respectively. Where using a cluster of eight noeds, the program scalability saturates at 36 and 40 cores for lammps and socorro respectively.

There is another close work done to model and predict the cloud system for running HPC applications without using the queueing networks; Shi et al. [14] introduce an instrumentation assisted complexity analysis methodology for program scalability analysis. Their methodology is based on Amdahl's and Gustafson's laws. Their work is re-implemented on the same experimental setup described in this paper for fair comparison. Two kernels of NASA parallel benchmark suite are used in this comparison; CG and MG with class-C problem size for both. The prediction of their model and the proposed model against the actual measured execution time are shown in Fig. 4. Their model does not count for the heterogeneity in the communication operations in parallel applications where there are inter- and intra- communication operations among the working processes. Their model counts only for intercommunication operations among the processing units not the intra-communication operations occurred inside the same processing unit. So, the accuracy for both models are almost the same for the experiments where every server in the working cluster contains only one process; up to 8 servers. The average accuracy for both kernels was 89% for the proposed model where the scalability analysis model achieves 87.3% accuracy. However, accounting for the intra-communication operations with no resources contention (using the available number of cores only), the proposed model results in 87.4% average accuracy for both kernels preserving the same level of accuracy, where the scalability analysis model's accuracy degrades to only 61%.

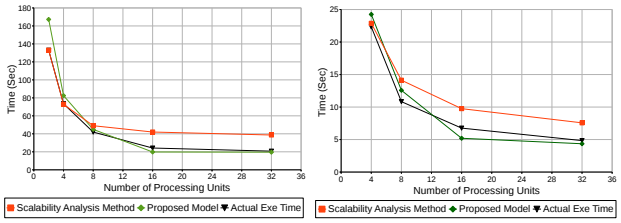
6 CONCLUSION AND FUTURE WORK

In this paper we propose an analytical performance model for predicting the performance of HPC MPI applications on the cloud bare-metal multicore servers. The model considers the contention



(a) Cluster of two machines. **(b) Cluster of four machines.** **(c) Cluster of eight machines.**

Figure 3: Running time prediction results of SPEC-MPI benchmarks; Lammps and Socorro on three different size clusters.



(a) CG benchmark. **(b) MG benchmark.**

Figure 4: Proposed model running time prediction results of NAS Parallel Benchmarks; CG and MG compared to the scalabilty analysis method prediction.

on both computation and communication resources through modelling them as a queueing network. In addition, the model accounts for the heterogeneity with the cloud bare-metal servers, where the model considers the processors speed and available number of cores (which varies among servers) as parameters as well as considering both the intra and inter communication operations between processes inside the same server or among different servers. For our experiments, we consider a cluster of multiple bare-metal cloud servers as the underlying system and two benchmarks from the SPECMPI suite as well as two kernels from NASA Parallel benchmarks suit as the workload. The results show a prediction, with 86% average accuracy, to the execution times of the two running benchmarks for different configurations of compute nodes; two, four and eight different machines.

Thus the model can potentially be used to assess the cost of resource usage on the performance of the cloud physical machines with different virtual to physical configurations, and to aid in developing better schedulers, which is an important area for future work. In particular, the queueing service times can account for the hypervisor overhead. Also, the number of working processes on each processing node would reflect the number of working virtual machines on each physical host, allowing for modelling resource sharing contention. Moreover, we need to extend the model to account for interference with other workloads running simultaneously on the same multicore machines.

REFERENCES

[1] Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis de Supinski, and Martin Schulz. 2008. A Regression-based Approach to Scalability Prediction. In *Proceedings of the 22Nd Annual International Conference on Supercomputing (ICS '08)*. ACM, New York, NY, USA, 368–377. https://doi.org/10.1145/1375527.1375580

[2] Patrick G. Bridges, Matthew G. F. Dosanjh, Ryan Grant, Anthony Skjellum, Shane Farmer, and Ron Brightwell. 2015. Preparing for Exascale: Modeling MPI for Many-core Systems Using Fine-grain Queues. In *Proceedings of the 3rd Workshop on Exascale MPI (ExaMPI '15)*. ACM, New York, NY, USA, Article 5, 8 pages. https://doi.org/10.1145/2831129.2831134

[3] NASA Advanced Supercomputing Division. [n. d.]. NASA Parallel Benchmarks Suite. https://www.nas.nasa.gov/publications/npb.html.

[4] Ifeanyi P Egwutuoha, Shiping Chen, David Levy, and Rafael Calvo. 2013. Cost-effective Cloud Services for HPC in the Cloud: The IaaS or The HaaS?. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. 217.

[5] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. 2003. *Introduction to Parallel Computing*. Pearson Education Limited, Chapter Parallel Programming Platforms, 58–60.

[6] Abhishek Gupta, Laxmikant V. Kalé, Dejan S. Milojicic, Paolo Faraboschi, Richard Kaufmann, Verdi March, Filippo Gioachin, Chun Hui Suen, and Bu-Sung Lee. 2012. Exploring the Performance and Mapping of HPC Applications to Platforms in the Cloud. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '12)*. ACM, New York, NY, USA, 121–122. https://doi.org/10.1145/2287076.2287093

[7] J. Li, M. Humphrey, C. van Ingen, D. Agarwal, K. Jackson, and Y. Ryu. 2010. eScience in the cloud: A MODIS satellite data reprojection and reduction pipeline in the Windows Azure platform. In *IPDPS 2010*. 1–10. https://doi.org/10.1109/IPDPS.2010.5470418

[8] Aniruddha Marathe, Rachel Harris, David K. Lowenthal, Bronis R. de Supinski, Barry Rountree, Martin Schulz, and Xin Yuan. [n. d.]. A Comparative Study of High-performance Computing on the Cloud. In *HPDC '13*.

[9] M. A. Netto, R. L. Cunha, and N. Sultanum. 2015. Deciding When and How to Move HPC Jobs to the Cloud. *Computer* 48, 11 (Nov. 2015), 86–89. https://doi.org/10.1109/MC.2015.351

[10] Paul Rad, AT Chronopoulos, P Lama, Pranitha Madduri, and Cameron Loader. 2015. Benchmarking bare metal cloud servers for HPC applications. In *Cloud Computing in Emerging Markets (CCEM), 2015 IEEE International Conference on*. IEEE, 153–159.

[11] Lavanya Ramakrishnan, Keith R. Jackson, Shane Canon, Shreyas Cholia, and John Shalf. 2010. Defining Future Platform Requirements for e-Science Clouds. In *SoCC (SoCC '10)*. ACM, NY, USA, 101–106. https://doi.org/10.1145/1807128.1807145

[12] M. Reiser and S. S. Lavenberg. 1980. Mean-Value Analysis of Closed Multichain Queuing Networks. *J. ACM* 27, 2 (April 1980), 313–322. https://doi.org/10.1145/322186.322195

[13] A. Saad and A. El-Mahdy. 2013. Network Topology Identification for Cloud Instances. In *2013 Int. Conf. on Cloud and Green Computing*. 92–98. https://doi.org/10.1109/CGC.2013.22

[14] J. Y. Shi, M. Taifi, A. Pradeep, A. Khreishah, and V. Antony. 2012. Program Scalability Analysis for HPC Cloud: Applying Amdahl's Law to NAS Benchmarks. In *SC12*. 1215–1225. https://doi.org/10.1109/SC.Companion.2012.147

[15] The Standard Performance Evaluation Corporation (SPEC). [n. d.]. SPEC MPI 2007 Benchmark Suite Documentation. https://www.spec.org/auto/mpi2007/Docs.

[16] Guohui Wang and T. S. Eugene Ng. [n. d.]. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *INFOCOM'10*.

[17] Yong Wang. 2012. GaussâĂŞNewton method. *Wiley Interdisciplinary Reviews: Computational Statistics* 4, 4 (7 2012), 415–420. https://doi.org/10.1002/wics.1202

[18] Y. Zhai, M. Liu, J. Zhai, X. Ma, and W. Chen. 2011. Cloud versus in-house cluster: Evaluating Amazon cluster compute instances for running MPI applications. In *SC11*. 1–10.

A Design Space Exploration Tool Set for Future Tera-scale High-Performance Computers

Roberto Giorgi, Marco Procaccini and Farnam Maybodi Khalili

A Design Space Exploration Tool Set for Future 1K-core High-Performance Computers

Roberto Giorgi
Department of Information Engineering and Mathematics, University of Siena
giorgi@diism.unisi.it

Marco Procaccini
Department of Information Engineering and Mathematics, University of Siena
procaccini@diism.unisi.it

Farnam Khalili
Department of Information Engineering and Mathematics, University of Siena
Department of Information Engineering, University of Florence
khalili@diism.unisi.it

ABSTRACT

Given the constantly growing complexity of multi-core architectures, Design Space Exploration (DSE) tools play an important role to evaluate different design options. In this paper, we present a DSE toolset targeting massively parallelized HW/SW architectures with a high degree of flexibility in order to successfully simulate multi-core-multi-node platforms. Our DSE tools provide a rapid and simple-to-use work-flow to easily retrieve and analyze the key metrics and eventually evaluate the design. We examine the DSE toolset and methodology while performing several simulations of a general purpose 1K-core architecture and evaluate not only standard metrics like the L2 cache miss rates, but also operating system activity and its impact. We leverage the knowledge gained in our methodology to develop and evaluate a novel dataflow execution model named "DataFlow-Threads" (DF-Threads). We validated the outcomes of the simulator against an equivalent FPGA-based design.

Keywords

Design Space Exploration; Simulation; Performance Analysis and Design; Multi-Core

1. INTRODUCTION

Recently, in order to match the performance request with the design requirement, researchers more than ever rely on the heterogeneous and domain-specific architectures [24]. Future architectures may be composed of thousands of tightly coupled cores (CPUs and GPUs), residing nearby accelerators, and become more complex than current ones [3]. Moreover, modern embedded systems are increasingly based on heterogeneous Multi-Processor SoC (MP-SoC) architectures. To cope with the design complexities of such architectures, Design Space Exploration (DSE) is an important portion of the design flow. DSE and its automation is a significant part of modern performance evaluation and estimation methodologies in order to reduce the design complexity, time-to-market, and find the best compromise among design constraints in respect to the application. Computer designers, therefore, rely on simulations as part of the DSE work-flow to perform early-stage design assessments in order to save time and costs. A simulator not only ensures the functional correctness but also may provide an accurate timing information.

We started to develop DSE tools within the TERAFLUX project [7] in order to evaluate a complex architecture with e.g., 1000 general purpose cores and running a full OS like Linux. In this paper, we present a set of DSE tools and experiments that we made by the COTSon simulator during the AXIOM project [1, 8, 9, 12, 29, 30] and the TERAFLUX project. Our DSE tools permit us to model features of architectures which are not yet available on the market in a rapid way, we were able to easily evaluate several execution models such as Cilk, OpenMPI, and a novel execution model like DataFlow-Threads (DF-Threads) [14, 18, 31] either for embedded [11] or for multi-core [10] systems.

The remainder of this paper is organized as follows: in Section, 2 we discuss the important challenges and issues regarding the exploration of an architecture; in Section, 3 we outline our DSE toolset and present our simulation platform; in Section, 4 we evaluate our DSE through different types of test case and discuss the evaluation results; in Section, 5 we highlight some related DSEs and scholar evaluation platforms, and finally, we conclude the paper in Section 6.

2. PROBLEM STATEMENT

Evaluation of a multi-core architecture even at the prototype stage is quite challenging, time-consuming. Moreover, it is not always possible to get the "perfect" setup, and hardware prototyping possibly imposes several limitations. In this section, we briefly highlight these limitations and show the importance of simulator (like COTSon [2]) when it comes to assess and retrieve key metrics of a high-performance computer, e.g., 1000 general purpose cores.

We report in Table 1 different approaches like using a physical Cluster, FPGA, and Simulator to evaluate and do research related to 1000-core computing system (Information revised from the RAMP project [6]).

Given a cluster at a scale of, e.g., 1000 general purpose

RAPIDO '19, January 21–23, 2019, Valencia, Spain

ISBN 978-1-4503-6260-3/19/01…$15.00

DOI: https://doi.org/10.1145/3300189.3300195

Table 1: comparison different approaches for evaluating large computing system. the grade points are scaled between 0 and 5 (grade of 5+ implies the superiority); GPA: Grade Point Average

	Cluster	FPGA	Simulator
Scalability	5	5)	5
Cost	3	4(€0.1-0.2M)	5+(€0.01M)
Observability	3	5+	5+
Reproducibility	2	5+	5+
Reconfigurability	3	5+	5+
Credibility	5+	3.5 to 4.5	3
Development time	4	3	5+
Performance (clock)	5(3GHz)	1(GHz)	3
Modifiable	0	4	5
GPA	3.38	3.2 to 3.7	4.8

cores, the best possible solution to connect the nodes (each node may consist of several cores) could be through InfiniBand interconnect. The main disadvantage of a physical cluster is its high cost and its inflexibility towards modification of architecture as well as poor extensibility in order to reconfigure the Instruction Set Architecture (ISA).

For FPGA, the hardware and software must be configured and set up, which invokes considerable time and also effort.

The simulators might not show satisfying credibility, but as they evolve, their credibility also improves. The main problem of simulators is their less performance in comparison with a physical cluster. However, we consider the simulators very useful for approaching a reasonable level of accuracy, scalability and simulation speed. Importantly, COTSon [2] offers a flexible simulation environment which made possible to design our DSE, and add new instructions [19, 25, 26] in order to evaluate a multi-core architecture, with a dataflow execution model like DF-Threads.

We use COTSon to offer a flexible DSE toolset that easily can adopt new hardware/software platforms, and support scalability for a multi-node architecture. For instance, in order to address the challenges of a 1k-core architecture, we should be able to have a full-system simulation including Operating System (OS), application benchmarks, a memory hierarchy and all peripherals as well.

3. SIMULATION FRAMEWORK

Our proposed framework allows us to modify system parameters such as the number of cores and number of simulated instances (nodes), which are running in parallel on completely independent hosts. This framework is also able to run Shared Memory application like OpenMP as well.

The proposed simulation framework relies on HP-Labs COTSon simulation environment and on a set of customized tools that are intended to easily setup the experimental environment, run experiments, extract and analyze the results.

3.1 The COTSon simulator

COTSon simulator [2] is based on the so-called "functional-directed" approach, where the functional execution is decoupled from the timing feedback. COTSon simulator uses the AMD SimNow virtualizer tool, which is proposed by AMD in order to test and develop their processors and platforms. COTSon executes its functional model into the SimNow virtual machine, running and testing the execution of the functional model. A custom interface is provided, in order to facilitate the exchange of the data between COTSon and the internal state of SimNow. As can be seen in Figure 1, COTSon architecture is made of three main components:

1) FUNCTIONAL MODELS: it contains the instances of the SimNow virtualizer, which executes the functional model based on a configurable x86-64 dynamically translating instruction-level platform simulator.
In fact, we were able to customize the x86-64 instruction set of SimNow in order to introduce new instructions for the implementation of the DF-Threads execution model [19].

2) TIMING MODELS: it implements simulation acceleration techniques, such as dynamic sampling, the tracing, profiling and statistics collection. Through the specification of a timing model for a given component (i.e., L1 cache, networking), we can model different behaviors. The timing models are decoupled from the functional execution of SimNow, allowing us the flexibility to model different types of architectural feature.

3) SCRIPTING GLUE: the final part is related to the scripts used to boot/resume/stop each virtual machine, the setup of the parallel simulation instances of SimNow and the time synchronization among all the virtual machines.

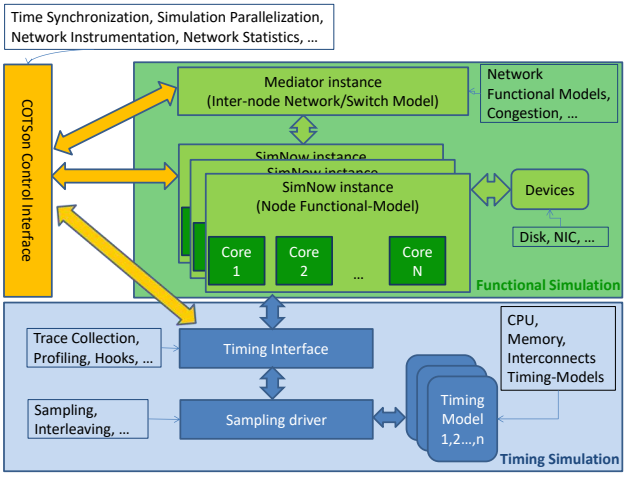


Figure 1: The COTSon simulation framework architecture.

3.2 Design Space Exploration Tools

3.2.1 *The tool set*

In order to guarantee a proper scientific methodology for experimentation, we developed the Design Space and Exploration Tools (DSE Tools) through which is possible to easily set up the COTSon simulation environment, extract and analyze the results of the experiments.
The normal toolflow is to follow the next eight steps.

i) GENIMAGE: the SimNow virtual machine needs a hard-drive image, which contains the Operative System to run. The GENIMAGE tool has the goal to create a customized version of a Linux distribution by other tools like VMBuilder and Debootstrap [5].

ii) ADD-IMAGE: this tool is preparatory to the GENIMAGE tool and it serves to load a given hard-drive image and the related virtual machine snapshot.

iii) BOOTSTRAP: it is a preparatory tool to prepare a user-based COTSon installation. This tool aims at solving the dependencies needed by the toolset in the host machine, avoiding the manual installation of them. It requires root permission once per machine. Moreover, the tool tunes some kernel parameters such as the number of host memory pages needed by SimNow.

iv) CONFIGURE: it enables multiple users on the host machine to run a configuration of their own simulation setup without the need of system administrator intervention. In fact, the tool runs completely in user-space, without the need of root permissions.

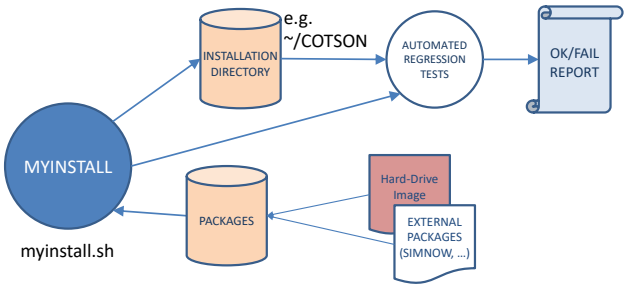


Figure 2: TOOLFLOW for the MYINSTALL tool. MYINSTALL prepares the whole environment for simulation-based Design Space Exploration with a single command. The Configuration File specifies which additional tool or tool-options should be used (e.g., non-public tools, or tools under NDA).

v) MYINSTALL: the purpose of this tool is to facilitate the installation process of the simulator and the hard-disk image which contains the pre-selected Operative System that will runs into the SimNow machine (see Figure 2). Moreover, MYINSTALL allows the choice of the simulation software version, in order to enable more versions of the simulator to co-exist for regression test. Finally, the tool performs several regression tests at the end of the installation phase, in order to verify the software is correctly patched, compiled and installed. The entire process is completely automatic and it can be easily repeated on multiple and parallel simulation hosts.

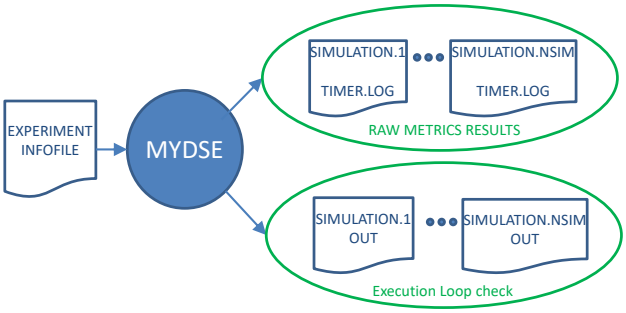


Figure 3: TOOLFLOW for the MYDSE tool. The experiment INFOFILE defines the simulation points and output files generated during each simulation are organized in order to facilitate their accessing and parsing by the other tools.

vi) MYDSE: we found a substantial need to implement a specific tool, which is able to easily catch possible failures or errors and, mostly, the automatic management of experiments in case of a large number of design points to be explored. As depicted in Figure 3, MYDSE relies on a small configuration file, named "INFOFILE", which is described in more details in the subsection 3.2.2.
Also, the tool is able to spread the simulation among multiple hosts and,if necessary, it can use the same binary with different GLIBC library version across the hosts. This allows us to use different operating systems, with a different version of the GLIBC library, in different guests. During the experiment, MYDSE controls the simulation loop, collecting in an ordered way the several files from each simulation point. Statistics based on user formulas are printed out at the end of each simulation, in order to provide an overall evaluation of the results. In case of failures, the tool kills the failed simulation and the related processes after a certain time, trying the re-execution of the failed simulation automatically. The timeout is derived by a simulation estimation model (i.e., proportional to the number of nodes and cores of the system).

vii) GTCOLLECT: once a campaign of experiments has been concluded, we need to collect, analyze and plot results in a simple way. In this perspective, we can extract data from experiments with the GTCOLLECT tool (GT stand for Graphic Table Collect), which prints out the collected data, based on the "INFOFILE" information and a "LAYOUT" text files where the user can specify the relevant output metrics to select. Furthermore, some additional calculations are performed on the data, such as the Coefficient of Variation, in order to analyze the correctness of the results.
With the GTCOLLECT tool, we can perform a complete analysis of the raw data produced by the MYDSE.

viii) GTGRAPH: once the results are collected in the GTCOLLECT format, the GTGRAPH tool can produce a graphical view of the data, like Figure 7,8,9.

Additionally, COTSon permits a connection with McPAT [2] to analyze the power consumption and the temperature of an experiment.

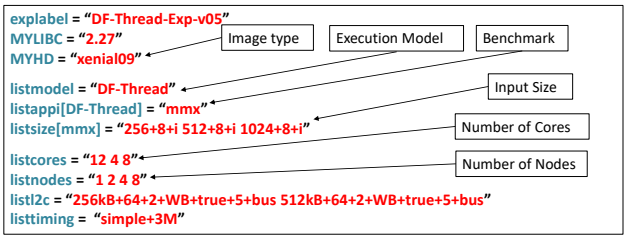


Figure 4: INFOFILE example, which describes a Design Space Exploration experiment.

3.2.2 *Experiment Description*

In this section, we want to introduce the experiment description file, named "INFOFILE", which makes the DSE easy to manage a clear identification of the Design Space. As depicted in Figure 4, we can describe the experiment

through a simple file that uses "Bash syntax": <variable> = "<string>". Each DSE variable is defined with the prefix "list", while "<string>" represents a set of value where elements can be separated with the character "+" (i.e. 256+8+i represent matrix size = 256, block size = 8 and matrix element type = integer). Moreover, we can define multiple configurations of the architecture, in order to find the best organization for a given application.

As we can see in Figure 5, each architecture configuration is composed of high-level blocks and we can specify the organization using the bash syntax of the INFOFILE. The names identify a block and the "main" block is the root of the configuration. The "-" character specifies the link between two blocks and the "+" character separates the different links. The "." character separates a first part which represents a single instance of the implicitly defined architectural blocks and a second part which represents multiple instances of the implicitly defined architectural blocks. For example, in the "listarch" variable of Figure 5 the part *.ic-cpu+busT-t2+l2-ic+l2-dc+t2-it+t2-dt* will be instantiated C times (where C is the number of Cores). Also, there is the possibility to insert a tracing module between two blocks of the architecture, snooping the data and the information exchanged among the blocks ("trace" block in Figure 5).

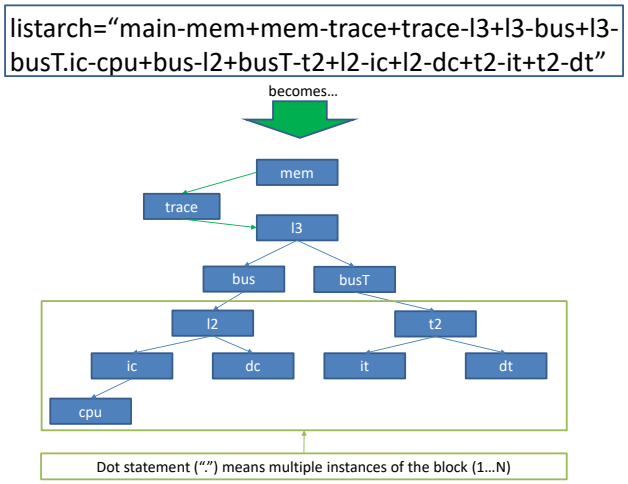


Figure 5: Architecture specification using the INFOFILE syntax, which it is used by the MYDSE tool to configure the COTSon simulation framework.

3.2.3 *Customizable devices*

The SimNow tool allows us to personalize the device architecture of the virtual machine that we want to run, by selecting a device from a list and placing it into the device tree. Moreover, the device list could be extended in order to introduce new customizable devices. As depicted in Figure 6, we were able to introduce a new PCI device, named XNIC, in order to emulate the behavior of a hardware accelerator device (e.g. FPGA) and at the bottom we show the kernel boot of such device.

4. EXAMPLE OF EVALUATIONS

In this section, we want to show some experiments that we made on the COTSon simulator during the AXIOM and the TERAFLUX European projects. The evaluations that we

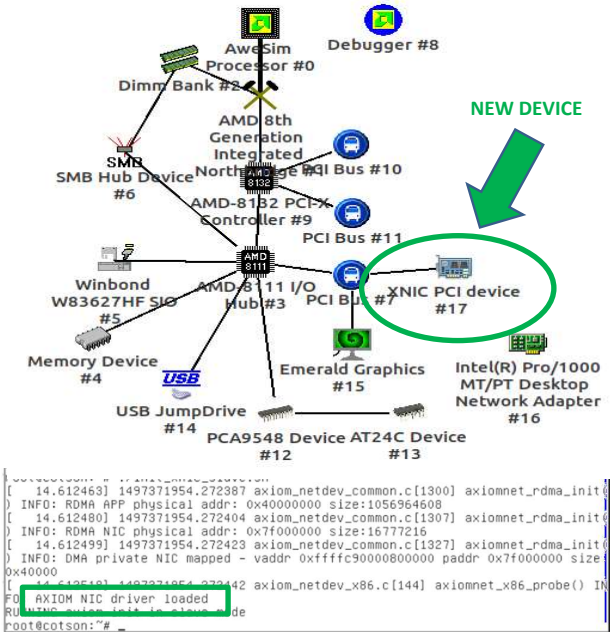


Figure 6: Architecture configuration of a SimNow virtual machine with the additional custom PCI device named XNIC. In the bottom part, the new device is loaded into SimNow.

show in this section are based on the DF-Threads execution model, focusing on execution time, OS impact and cache usage (we don't show results about temperature and power consumption because it is out of the scope of this paper).

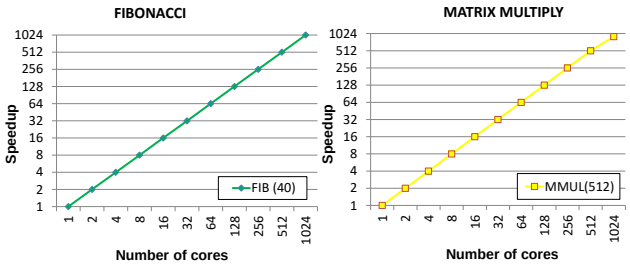


Figure 7: Mutli-node, multi-core simulation up to 1024 cores using Fibonacci, with input set to 40, and Matrix Multiply with matrix size 512x512 based on DF-Threads execution model.

TEST CASE 1: as can be seen in Figure 7, we tested the DF-Threads execution model through two well-known benchmarks like Fibonacci (input n=40) and Matrix Multiply (with a matrix size of 512). We are able to simulate different nodes/cores configuration, from 1 to 1024 cores. Each node is configured to have from 1 to 32 cores and the node range is from 1 to 32.

TEST CASE 2: thanks to our DSE tools, we were able to study the scaling factor while varying the input size and the number of nodes, demonstrating that the DF-Threads execution model has good scaling in every tested Operating System. We tested the performance of different OS distribution with a different size of the Matrix Multiplication benchmark. As depicted in Figure 8, the performance vary as the OS distribu-

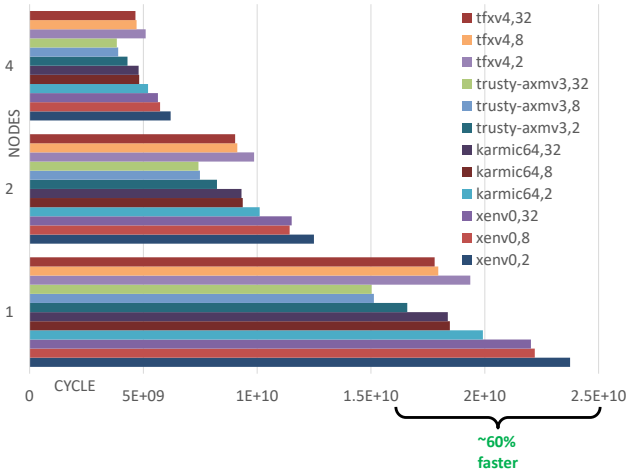


Figure 8: Study of the variation in performance (execution cycles) in the different OS distribution as we vary the L2 Cache size from 2KiB to 32KiB. The benchmark is Matrix Multiply with matrix size of 512 . Kernel activity is responsible from 10% to 50% on the execution time.

tion is changed and the trusty-axmv3 seems to be the best among the tested ones in most cases. According to [11], the Kernel activity has a huge impact on the performance of the different operating system, varying from 10% to 50%.

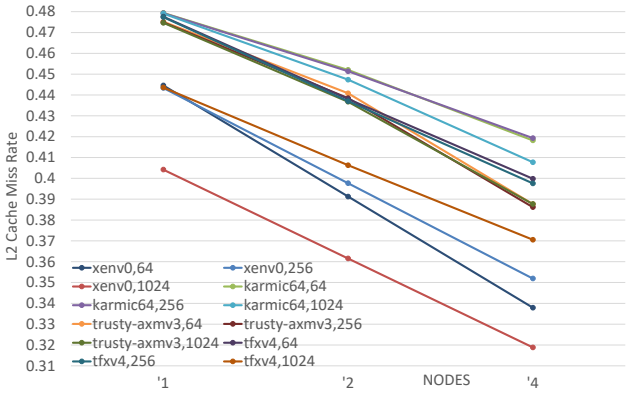


Figure 9: Example metric (L2 Cache Miss Rate). We vary the number of nodes (from 1 to 4), the OS distribution and the input size of the Matrix Multiply benchmark (matrix size = 256,512,and 1024).

TEST CASE 3: in this test we want to exploit the output traces of the simulator in order to analyze different aspects of the experiments. As we can see in Figure 9, we study the L2 cache miss rate behavior among the different OS distribution. We choose the input size at 512x512 in order to have enough parallelism and we vary the L2 cache size from 32 to 1024 KiB. The results show that the L2 cache size has a strong influence in the performance and therefore it may play a crucial role in the system. For example, we discovered that one of the optimization points of the execution model implementation should be the L2 cache usage.

VALIDATION TEST: Finally, we validated our results obtained through the utilization of the simulator, comparing the execution time (sequential execution) of the Matrix Multiplication Benchmark both in simulator and on a real board FPGA based (AXIOM-Board [9, 15, 17, 30]). As can be seen in Figure 10, the results of the simulator and the board are very close, confirming our performance predictions, despite some architectural differences.

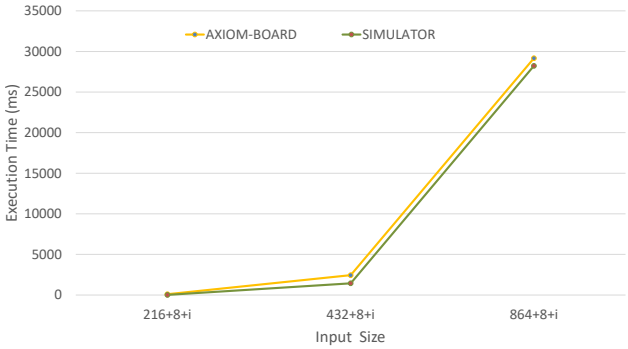


Figure 10: Preliminary comparison of the execution time between the simulator and the AXIOM-Board. We used the Matrix Multiply benchmark three different size: 216,432 and 864.

5. RELATED WORK

Several types of research have been recently carried out and focused on Design Space Exploration (DSE) and simulators like MULTICUBE [28], which proposed a multi-level simulation approach using an approximate analytic metamodel for MP-SoC architectures based on Artificial Neural Networks. An iterative multi-objective optimization technique for MP-SoC is proposed in [20], which is able to model the correlation of the multi-processor configurations with appropriate analytical functions. In order to accurately simulate more complex processor designs, conventionally, architectural simulators like Trace-Factory [16] and GEMS [21] have been adopted. GEMS provides a very detailed and accurate simulation by using a timing first approach to model timing. In our case, we relay on COTSon, which uses functional directed approach allowing simulation of several thousands full system cores. GRAPHITE [22] is a multi-core simulator designed to support more than 1000 cores at higher-level of abstraction using distribution workload techniques. These simulators lack efficient parallelization capabilities due to fine-grained synchronization difficulties, which limit the speedup. To mitigate these limitations, the user has to trade speed with accuracy [22, 23].

Sniper [4] is a parallel and scalable multi-core simulator which compromises between accurate high-abstraction analytical models and fast parallel simulation, and covers a larger portion of the hardware design space. Even though it covers operating system runtime simulations, they simulate only up to 16-cores, while in our case we successfully are able to simulate a 1000 general purpose multi-core architecture [10]. However, some of these approaches miss portability and require considerable time to be ported to other frameworks.

In [25] authors propose a full stack simulation system targeting heterogeneous kilo-core architectures which includes a customized extended version of x86/64 ISA [19,26] to sup-

port DataFlow-Threads (DF-Threads) execution model [14]. In [13,27] COTSon [2] simulator has been leveraged in order to provide Distributed Scheduler to support many-node architectures, and significantly improve scalability and power estimation.

6. CONCLUSION

We have presented a set of tools for the Design Space Exploration, based on the COTSon simulator framework, with the aim of supporting large set of experiments of a multi-node multi-core platform with full OS execution (e.g., 1000 general purpose processors and real OS activity). Thanks to our tools, we setup the simulation environment in less than ten minutes, including several regression tests, saving hours in comparison with manual installation. We can run several experiments by using a simple configuration file, handle possible failures or errors during the simulations. Finally, results are automatically collected and presented in the desired graphical view.

We showed several test cases and a validation test against a FPGA platform. The results permitted us to derive information early in the design process.

The DSE tools that we presented in this paper were massively used during two European projects (TERAFLUX and AXIOM), facilitating the exploration of a large design space and the test of a new execution model (DF-Threads).

In the future, we want to exploit machine learning techniques to better select design points and in order to improve the statistical characterization of the collected data.

7. ACKNOWLEDGMENT

The authors would like to thank Stefano Viola of SECO (s.r.l) for his support in developing the XNIC PCI device, The European Commission under the AXIOM H2020 project (id. 645496), TERAFLUX (id. 249013), and HiPEAC (id. 779656), and importantly, the anonymous reviewers for their helpful comments.

8. REFERENCES

[1] C. Alvarez et al. The AXIOM software layers. *ELSEVIER Microprocessors and Microsystems*, 47, Part B:262–277, 2016.

[2] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. COTSon: infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, 2009.

[3] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[4] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 52. ACM, 2011.

[5] Debootstrap website. https://wiki.debian.org/Debootstrap.

[6] G. Gibeling, A. Schultz, and K. Asanovic. The ramp architecture & description language. In *2nd Workshop on Architecture Research using FPGA Platforms*, 2006.

[7] R. Giorgi. Teraflux: Exploiting dataflow parallelism in teradevices. In *ACM Computing Frontiers*, pages 303–304, Cagliari, Italy, May 2012.

[8] R. Giorgi. Scalable embedded systems: Towards the convergence of high-performance and embedded computing. In *Proc. 13th IEEE/IFIP Int.l Conf. on Embedded and Ubiquitous Computing (2015)*, pages 148–153, Oct. 2015.

[9] R. Giorgi. AXIOM: A 64-bit reconfigurable hardware/software platform for scalable embedded computing. In *6th Mediterranean Conf. on Embedded Computing (MECO)*, pages 113–116, June 2017.

[10] R. Giorgi. Exploring future many-core architectures: The TERAFLUX evaluation framework. In *Advances in Computers*, Advances in Computers, pages 33–72. Elsevier, 2017.

[11] R. Giorgi. Scalable embedded computing through reconfigurable hardware: comparing df-threads, cilk, OpenMPI and jump. *ELSEVIER Microprocessors and Microsystems*, 63:66–74, Aug. 2018.

[12] R. Giorgi, N. Bettin, P. Gai, X. Martorell, and A. Rizzo. *AXIOM: A Flexible Platform for the Smart Home*, chapter 3, pages 57–74. Springer Int.l Pub., Cham, 2016.

[13] R. Giorgi et al. TERAFLUX: Harnessing dataflow in next generation teradevices. *Microprocessors and Microsystems*, 38(8, Part B):976–990, 2014.

[14] R. Giorgi and P. Faraboschi. An introduction to DF-Threads and their execution model. In *IEEE MPP*, pages 60–65, Paris, France, Oct. 2014.

[15] R. Giorgi, F. Khalili, and M. Procaccini. Energy efficiency exploration on the zynq ultrascale+. In *The 30th International Conference on Microelectronics (ICM)*, December 2018.

[16] R. Giorgi, C. Prete, G. Prina, and L. Ricciardi. Trace factory: Generating workloads for trace-driven simulation of shared-bus multiprocessors. *IEEE Concurrency*, 5(4):54–68, Oct. 1997.

[17] R. Giorgi, M. Procaccini, and F. Khalili. AXIOM: A scalable, efficient and reconfigurable embedded platform. In *Design, Automation and Test in Europe, the european event for electronic system design and test (DATE)*, March 2019.

[18] R. Giorgi and A. Scionti. A scalable thread scheduling co-processor based on data-flow principles. *Future Generation Computer Systems*, 53:100–108, Dec. 2015.

[19] N. Ho, A. Portero, M. Solinas, A. Scionti, A. Mondelli, P. Faraboschi, and R. Giorgi. Simulating a multi-core x86-64 architecture with hardware isa extension supporting a data-flow execution model. In *IEEE Proc. AIMS-2014*, pages 264–269, Madrid, Spain, Nov. 2014.

[20] G. Mariani et al. A correlation-based design space exploration methodology for multi-processor systems-on-chip. In *DAC*, pages 120–125. ACM, 2010.

[21] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[22] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[23] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, M. D. Hill, D. A. Wood, S. Huss-Lederman, and J. R. Larus. Wisconsin wind tunnel ii: a fast, portable parallel architecture simulator. *IEEE*, 8(4):12–20, 2000.

[24] D. Patterson. 50 years of computer architecture: From the mainframe cpu to the domain-specific tpu and the open risc-v instruction set. In *Solid-State Circuits Conference-(ISSCC), 2018 IEEE International*, pages 27–31. IEEE, 2018.

[25] A. Portero et al. Simulating the future kilo-x86-64 core processors and their infrastructure. In *45th Annual Simulation Symp. (ANSS12)*, pages 62–67, Orlando, FL, Mar 2012.

[26] A. Portero, Z. Yu, and R. Giorgi. T-star (t*): An x86-64 isa extension to support thread execution on many cores. In *HiPEAC ACACES-2011*, pages 277–280, Fiuggi, Italy, July 2011. poster.

[27] A. Portero, Z. Yu, and R. Giorgi. Teraflux: Exploiting tera-device computing challenges. *ELSEVIER Procedia Computer Science*, 7:146–147, 2011. Proc. 2nd European Future Technologies Conf. and Exhibition 2011 (FET 11).

[28] C. Silvano et al. Multicube: Multi-objective design space exploration of multi-core architectures. In *IVLSI*, pages 488–493. IEEE, 2010.

[29] D. Theodoropoulos et al. The AXIOM project (agile, extensible, fast i/o module). In *IEEE Proc. 15th Int.l Conf. on Embedded Computer Systems: Architecture, MOdeling and Simulation*, pages 262–269, July 2015.

[30] D. Theodoropoulos et al. The AXIOM platform for next-generation cyber physical systems. *ELSEVIER Microprocessors and Microsystems*, pages 540–555, 2017.

[31] L. Verdoscia and R. Giorgi. A data-flow soft-core processor for accelerating scientific calculation on FPGAs. *Mathematical Problems in Engineering*, 2016(1):1–21, Apr. 2016. article ID 3190234.