HIPEAC 2020 Conference

Proceedings of

RAPDIO 2020 Workshop

Bologna, Italy

21th January 2020

#HiPEAC20

January 20-22, 2020, Bologna, Italy

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

`https://rapidoworkshop.github.io/`

RAPIDO

Schedule

— **Workshop Introduction** 10 : 00 − 10 : 05

— **Session 1** 10 : 05 − 10 : 50
- — **Keynote 1 : Reda Nouacer**, CEA
  Digital twin for cyber physical systems

— **Session 2a** 11 : 30 − 12 : 15
- — **Keynote 2 : Fadi Kurdahi**, Center for Embedded & Cyber-physical Systems University of California, Irvine
  Towards Self-Aware Systems-on-Chip Through Intelligent Cross-Layer Coordination

— **Session 2b** 12 : 15 − 12 : 55
- — Boutheina Bannour and Arnault Lapitre
  Heuristic-aided Symbolic Simulation for Trickle-based Wireless Sensors Networks Configuration
- — Éder F. Zulian, Germain Haugou, Christian Weis, Matthias Jung and Norbert Wehn
  System Simulation with PULP Virtual Platform and SystemC

— **Session 3a** 14 : 00 − 14 : 45
- — **Keynote 3 : Muhammad Shafique**, Vienna University of Technology Institute of Computer Engineering
  Security for Machine Learning : The Intelligence Features of Your Smart Cyber Physical Systems are under Attack !

— **Session 3b** 14 : 45 − 15 : 25
- — Irune Yarza, Mikel Azkarate-Askatsua, Peio Onaindia, Philipp Ittershagen, Kim Grüttner and Wolfgang Nebel
  Static/Dynamic Real-Time Legacy Software Migration - A Comparative Analysis
- — WIP paper : Vincent Morice, Florence Maraninchi and Jérôme Cornet
  Towards A Power Advisor in a Devkit for Internet-of-Things Microcontrollers

— **Session 4** 16 : 00 − 16 : 45
- — **Keynote 4 : Eugenio Villar**, Grupo de Ingeniería Microelectrónica Universidad de Cantabria
  Mega-Modeling and Model-Driven Performance Analysis of CPSoS

List of regular papers

Keynotes

Keynote 1 **10 :05 - 10 :50**

Digital twin for cyber physical systems

— *Reda Nouacer, CEA*

— **Abstract :** Since the first use of the word "Digital-Twin" by Michael Grieves in a 2003, several research publications have addressed the technical obstacles underlying the implementation of this concept. In parallel with these academic research works, several industrial research projects have been undertaken for an evaluation in real or representative conditions of this approach. Certain sectors, such as nuclear and aeronautics, exploit digital twins throughout the life cycle of their products and in particular for the training of operating or maintenance personnel. Today the proof is made of the usefulness of the digital twin but unfortunately we note that the deployment of this approach is limited to a few areas of application and only the category of large industrial groups have access to it but in limited usages. The objective of the presentation is, firstly, to review the theoretical foundations and known uses of the digital twin in the field of CPS. Then to expose, in a second part, the blocking points to a massive deployment of this approach in the industrial world. The presentation ends with the proposal of the economic model MSaaS (Modeling and Simulation as a Service) which should answer a certain number of the identified problems and allow the emergence of an agile ecosystem to meet the needs of the industry in this field.

— **Biography :** Reda NOUACER is a research engineer at CEA LIST where he work on design space exploration and virtual platforms. Before he worked at Prosilog SA and then at Texas Instruments. His research interests include design space exploration, hardware simulation, and dependability using virtual platforms. He earned a HW/SW Engineer degree in 1993 and the Magister degree in 1997 in Computer Engineering from the Badji-Mokhtar University (Annaba-Algeria). His thesis entitled 'CAMELEON : A Parallel Architecture Emulator' summarizes his work on building a low-cost emulator of parallel architectures for parallel programs validation. Reda NOUACER is and has been involved in many interdisciplinary national and international basic research projects as well as industrial research projects.

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

Towards Self-Aware Systems-on-Chip Through Intelligent Cross-Layer Coordination

— *Fadi Kurdahi, Center for Embedded & Cyber-physical Systems University of California, Irvine*

— **Abstract :** Although there is a rich history of cross-layer design for embedded computing systems to achieve desired QoS, we are facing ever more challenges from the intertwined goals of energy- efficiency, thermal design constraints, as well as resilience to errors emanating from the application, environment and hardware platforms. We posit that next-generation computing platforms must necessarily deploy intelligent cross-layer design achieved through self-awareness principles inspired by biology and nature. Such an approach will move us from current strategies (using limited cross-layer coordination) to a holistic cross-layer strategy that enables intelligent cross-layer management policies which can adaptively tune itself based on the current state of the system. The talk will present design exemplars that embrace this intelligent cross-layer approach, and highlight the role of self-awareness in achieving dynamic adaptivity.

— **Biography :** Fadi Kurdahi received his PhD from the University of Southern California in 1987. Since then,he has been a faculty at the Department of Electrical & Computer Engineering at UCI, where he conducts research in the areas of Computer Aided Design and design methodology of large scale systems. He serves as the Associate Dean for Graduate and Professional Studies of the Henry Samueli School of Engineering, and the Director of the Center for Embedded & Cyber-physical Systems (CECS), comprised of world-class researchers in the general area of Embedded and Cyber-physical Systems. He served on numerous editorial boards, and was program chair or general chair on program committees of several workshops, symposia and conferences in the area of CAD, VLSI, and system design. He received the best paper awards for the IEEE Transactions on VLSI in 2002, ISQED in 2006 and ASP-DAC in 2016, and other distinguished paper awards at DAC, EuroDAC, ASP- DAC and ISQED. He also received the Distinguished Alumnus award from his Alma Mater, the American University of Beirut in 2008. He is a Fellow of the IEEE and the AAAS.

Keynote 3 **14 :00 - 14 :45**

Security for Machine Learning : The Intelligence Features of Your Smart Cyber Physical Systems are under Attack !

— *Muhammad Shafique, Vienna University of Technology Institute of Computer Engineering*

— **Abstract :** Access to massive amounts of data and high-end computers has heralded revolutionary advances in Machine Learning (ML) impacting domains ranging from autonomous driving and robotics, to healthcare, the natural sciences, the arts and beyond. As we deploy modern ML systems in safety- and health-care applications, however, it is important to ensure their security against adversarial attacks. Researchers have shown that many modern ML algorithms, especially the ones based on the deep neural networks (DNNs) are fragile and can be embarrassingly easy to fool. This is easier said than done. Recent research has shown that DNNs are susceptible to a range of attacks including adversarial input perturbations, backdoors, Trojans, and fault attacks. This can create catastrophic effects for various safety-critical applications like automotive, healthcare, etc. For instance, self-driving cars and vehicular networks, which heavily rely on ML-based functions, exhibit a wide attack surface that

can be exploited by well-known and yet-unknown-but-possible attacks on ML models. DNNs contain hundreds of millions of parameters and are hard to interpret/debug let alone verify, significantly increasing the chance they may misbehave. Further, any ML system is only as robust as the data on which we train it on. If the data distributions change in the field, this can impair performance (for example, an autonomous vehicle trained in day time conditions may not function at nighttime). The goal of this talk is to shed light on various security threats for the ML algorithms, especially the deep neural networks (DNNs). Various security attacks and defenses for DNNs will be presented in detail. Afterwards, open research problem and perspectives will be briefly discussed.

— **Biography :** Muhammad Shafique (M'11 - SM'16) received the Ph.D. degree in computer science from the Karlsruhe Institute of Technology, Germany, in 2011. He is currently a Full Professor with the Department of Informatics, Institute of Computer Engineering, TU Wien, Austria, where he is directing the group on Computer Architecture and Robust, Energy-Efficient Technologies. He holds one U.S. patent and over 200 papers in premier journals and conferences. His research interests include computer architecture, energy-efficient systems, robust computing, hardware security, brain-inspired computing, emerging technologies, and embedded systems. His research has a special focus on cross-layer analysis, the modeling, design, and optimization of computing and memory systems, and their integration in the Internet of Things and smart cyber-physical systems. He is a Senior Member of the IEEE and a member of the ACM, SIGARCH, SIGDA, SIGBED, and HiPEAC. He received the 2015 ACM/SIGDA Outstanding New Faculty Award, six gold medals, and several best paper awards and nominations at prestigious conferences. He served on the program committees of several conferences and gave several invited talks, tutorials, and keynotes.

Keynote 4 **16 :00 - 16 :45**

Mega-Modeling and Model-Driven Performance Analysis of CPSoS

— *Eugenio Villar, Grupo de Ingeniería Microelectrónica Universidad de Cantabria*

— **Abstract :** Model Based Design (MBD) has proven to be a powerful technology to address the development of increasingly complex embedded systems. Beyond complexity itself, challenges come from the need to target various execution platforms with different OSs and HW resources, even bare-metal, the increasing parallelism provided by them and its increasing heterogeneity. An additional difficulty comes from the tendency towards system applications in which the embedded system is only a piece of a much complex, distributed system. In any case, appropriate solutions improving performance, power consumption, cost, etc. have to be analyzed and selected. Addressing these challenges require flexible design technologies able to support from a single-source model its architectural mapping to different computing resources, of different kind and in different platforms. Thanks the potential of MBD methods and tools they should ensure flexibility and reusability. In this presentation, S3D, a UML/MARTE system modeling methodology is proposed able to address the challenges mentioned above by improving flexibility and scalability. This approach is illustrated and demonstrated on

a flight management system. The model is flexible enough to be adapted to different architectural solutions with a minimal effort by changing its underlying model of Computation and Communication (MoCC). Being completely Platform Independent, from the same model it is possible to explore and generate various solutions on different execution platforms.

— **Biography :** Prof. Eugenio Villar got his Ph.D. in Electronics from the University of Cantabria long time ago. Since 1992 is Full Professor at the Electronics Technology, Automatics and Systems Engineering Department of the University of Cantabria where he is currently the responsible for the area of HW/SW Embedded Systems Design at the Microelectronics Engineering Group. His research activity has been always related with system specification and modeling. His current research interests cover system specification and design, MpSoC modeling and performance estimation using SystemC and UML/MARTE of mixed-critical, distributed embedded systems. He is author of more than 130 papers in international conferences, journals and books in the area of specification and design of electronic systems. Prof. Villar served in several technical committees of international conferences like the VHDL Forum, Euro-VHDL, EuroDAC, DATE, VLSI-SoC, FDL, EuroMicro DSD and DAC. He has participated in several international projects in electronic system design under the FP5, FP6 and FP7, ITEA, Medea-Catrene, Artemis and ECSEL programs. He is the representative of the University of Cantabria in the ECSEL.

Heuristic-aided Symbolic Simulation for Trickle-based Wireless Sensors Networks Configuration

Boutheina Bannour and Arnault Lapitre

Heuristic-aided Symbolic Simulation for Trickle-based Wireless Sensors Networks Configuration

Boutheina Bannour, and Arnault Lapitre
CEA, LIST, Laboratory of Systems Requirements and Conformity Engineering
{boutheina.bannour,arnault.lapitre}@cea.fr

ABSTRACT

Wireless Sensor Networks (WSN) as parts of the so-called Internet of Things (IoT) are facing the challenge of upgrading their firmware very often, in particular when security flaws are found. This large-scale operation can be remotely conducted starting from frontier devices (or nodes) connected to the internet, and gradually spread among the rest of the network. The Trickle algorithm is an efficient well-known algorithm for versioned information dissemination in WSN, and is applicable for this kind of operations. The algorithm when well configured, allows: i) the reduction of the number of packets (messages) exchanged between devices to save their batteries, and ii) a quick propagation of new firmware versions to minimize periods during which the devices are outdated. In this paper, we develop model-based symbolic simulation for configuring Trickle-based WSN combined with an exploration heuristic to cope with combinatorial behavior of the network many-nodes, while still being able to highlight critical scenarios of outdated nodes' situations. Our simulation techniques are implemented in the tool Diversity, which shows promising usability and first results.

ACM Reference Format:
Boutheina Bannour, and Arnault Lapitre. 2020. Heuristic-aided Symbolic Simulation for Trickle-based Wireless Sensors Networks Configuration. In *Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '20), January 21, 2020, Bologna, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3375246.3375255

RAPIDO '20, January 21, 2020, Bologna, Italy

ACM ISBN 978-1-4503-7777-5/20/01…$15.00
https://doi.org/10.1145/3375246.3375255

1 INTRODUCTION

Context. Wireless Sensors Networks (WSN) allow remote monitoring based on the data they continuously collect, creating a direct connection between the digital systems and the physical world. They are components of typical applications of Internet of Things (IoT) in connected vehicles, household appliances, urban infrastructures, agriculture production, e-healthcare, etc. WSN are composed of many small devices, called nodes, that operate for long periods of time under the constraint of low consumption of energy, provided by very economical batteries. Like all IoT components, nodes connection to internet requires updating their firmware very often, in order to strengthen their functioning and security, e.g., against cyber-attacks such as the Mirai malware and its 27 emerging variants that have been discovered until recently, in 2019 [1]. The most recommended and easiest method of deployment is to use nodes connectivity to propagate firmware updates from some frontier nodes connected to the internet. But nodes communicate via short-range radio connections, and thus updates need to be gradually carried out from one node to neighbours until they are widespread in the network. This is reminiscent of the fact that the traffic must still be controlled and well distributed in the network so as not to exhaust those limited battery-powered nodes.

Motivation and related works. The Trickle algorithm [12, 14] continues to develop interest in low-power dissemination of versioned information in WSN. This algorithm is provided as a standard library in TinyOS [13] and Contiki [8], two well-known firmware Operating Systems (OS) for WSN. In addition, the Trickle algorithm is used in recently standardized WSN protocols namely the Multicast Protocol for Low Power and Lossy Networks (MPL) [9] and the IPv6 Routing Protocol for Low Power and Lossy Networks (RPL) [2]. The algorithm is an asset to the network traffic regulation as it allows neighboring nodes to quickly exchange new information if they are not consistent, and then, when they synchronize, suppress their transmissions until a new inconsistency is detected again. In a recent work [17], we have experimented test scenarios generation from MPL based on

sr-pairs coverage criterion [1]: We have observed a strong intertwining between the underlying (many) Trickle configuration(s) and the connectivity of the topology under test. With regard to given Trickle settings, some outdated nodes' situations have been more frequently observed when slightly modifying the topology connections (and vise versa). The algorithm is sensitive to the so-called redundancy constant which rules node transmissions a bounded number of times in case of inconsistency: since then, the node counts the number of times it hears from its neighbours the same (redundant) information as it holds, to stop the dissemination when the bound is reached. The choice of the redundancy constant, being global to all nodes, is somehow a tradeoff between reasonably distribute transmission load among nodes and not leaving some nodes outdated for long periods of time. It depends on the nature of applications and/or protocols, which employ Trickle: E.g., MPL [9] recommends using small values (values within 1-5) as it instantiates many Trickle instances per node (used often in dissemination of fragmented versioned firmware), and RPL [2] (used for routing purposes) recommends higher values (default value 10). Trickle uses two other bounds to dynamically adjust listening period of nodes to learn information from their neighbours, those are left as well to the appreciation of the expertise and/or with respect to the targeted topology characteristics: E.g., MPL recommends a lower bound of 10 times the expected connection-layer latency, usually corresponds to few milliseconds, and RPL suggests an upper bound of the order of few hours, etc. Recent works [5, 7, 11, 16, 21] have also pointed needs to carefully configure Trickle to benefit from its performance on reducing the network traffic while still being able to quickly reach an updated state of the network. Works in [5, 7, 11, 16, 21] propose analytical modelling of Trickle algorithm in order to study its performance in generic and/or non-uniform topologies. In particular, authors in [7] show that due to the unique global redundancy constant in Trickle, nodes with reduced connectivity transmit more often. They propose then a variant of Trickle relying on multiple local redundancy constants, each is a function of the number of neighbours of the node. All previously cited works use extensive numeric simulation and OS-based testbed experiments to validate their analyses.

Our approach and contribution. We propose symbolic execution [10] techniques to configure timed behaviors of many-nodes networks ruled by the Trickle algorithm. Unlike numeric simulation, symbolic execution animates or virtually executes models or code for symbolic input parameters rather than concrete values, which allows the exploration of many behaviours within a compact search space. In fact, each execution path is associated with logical constraints on those input parameters computed at each execution step, the so-called Path Conditions, those systematically identify equivalence classes of concrete behaviors encoded by the explored paths. We adopt a model-based approach in which models are asynchronous products of communicating Symbolic Timed Automata (STA) [20] over unbounded fifo-queue: We provide a generic Trickle pattern (as an STA) which can be re-used/tuned in modeling other applications or protocols based on the algorithm. In general, the exploration of such asynchronous models is quickly facing the combinatorial explosion problem [6]. So, we propose to combine a random search heuristic [3, 15] with nodes pairwise-connectivity analyses to improve the simulation of Trickle configurations. In particular, we show how those techniques can be used to efficiently highlight outdated nodes' situations, and hence adjust Trickle settings accordingly. We use the model-based symbolic execution tool Diversity [15] to implement and experiment our contributions.

Outline. Section 2 gives the necessary preliminaries on Trickle. Section 3 introduces our model-based symbolic simulation techniques for Trickle many-nodes networks, that we have implemented in the Diversity tool. Section 4 introduces our heuristic-aided configuration and evaluate it on different Trickle configurations, and Section 5 concludes the paper.

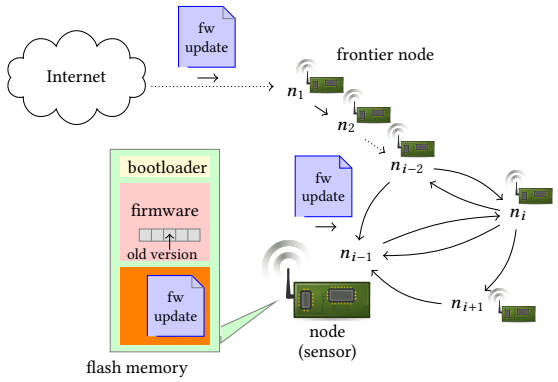

Figure 1: Updates in Wireless Sensors Network.

2 BACKGROUND ON TRICKLE

Trickle is a distributed algorithm that is executed by every node and can be summarized as follows [12, 14]:

- each node maintains a current interval τ, a counter c and a transmission time t in current interval τ,
- global parameters to all nodes are k the redundancy constant, τ_l (resp. τ_h) the smallest (resp. largest) value for τ,
- a node behaves according to following rules,

[1] In order to achieve full sr-pairs coverage every paired send-receive events must be executed at least once in testing [19].

(1) at the start of a new interval the node resets its timer and counter c and sets at random t to a value in $[\tau/2, \tau[$,
(2) if the node receives a message consistent with the information it holds, it increments c,
(3) when its timer reaches t, the node transmits the a message carrying the information it holds to all its neighbours only if c < k,
(4) when its timer expires at τ, it increases its interval length by setting τ to $\min(2 \cdot \tau, \tau_h)$ and starts a new interval,
(5) when a node receives a message that is inconsistent with its own information, then if $\tau > \tau_l$ it sets τ to τ_l and starts a new interval, otherwise it does nothing.

Each time an inconsistency is detected the current interval τ is reset, i.e., assigned with τ_l, and then it is doubled up to τ_h. When τ reaches τ_h it remains assigned with this value until the next inconsistency occurs. In a lossless network, the transmissions count per inconsistency is $\approx log(\tau_h/\tau_l)$: at most one transmission per τ-interval, occurring exactly at t. As illustrated in Figure 2, a node transmits only if its neighbours are unlikely to be up-to-date, when c < k given c counts receptions of consistent messages in the interval (k is global to all nodes). Now, if c reaches k, i.e., c ≥ k, the transmission at t is suppressed. We point out that the transmission time t is chosen at random within $[\tau/2, \tau[$ which imposes a listen-only period (first half of τ) for all nodes. The randomness of transmission time in favor of distributing the transmission load between nodes in the interval (and hence energy cost). When inconsistent information is received, small intervals are considered again, starting with the smallest τ_l, therefore nodes exchange quickly the information in their possession, with the objective to quickly synchronize nodes' information.

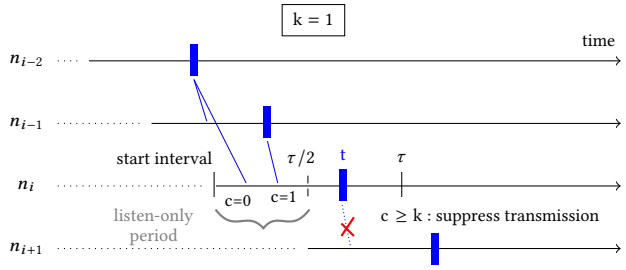

Figure 2: Trickle suppression mechanism.

3 SYMBOLIC SIMULATION

Symbolic models. For the purpose of modeling timed behavior of Trickle nodes, we will use the formalism of Symbolic Timed Automata (STA) [20]. Those are defined based on first-order logic and extend the well-known Timed Automata (TA) [4] with time guards using data-dependant bounds on clocks as being first-order terms (rather than being only bounded by constant values as in TA), data guards on other variables than clocks as first-order formulas, and updates of variables by substituting them by first-order terms. For instance, the transmission time will be constrained by the first-order formula clk = t, where clk is a clock which implements the Trickle timer, and t is indeed a data variable in symbolic domain $\tau/2 \leq t < \tau$, defined on its turn by the data variable τ being doubled at each new Trickle interval, ... This increased expressiveness of TA allows for compact models of (timed) behavior of Trickle-nodes as we will see in the rest of the section.

As part of our contribution, we have implemented in Diversity STA, besides, we have equipped them with a sequential statement language so that a single execution step can cover many intermediate computations. Those statements can include guards (formulas built on system clocks and data variables), communication actions involving systems ports (reception and storage on variables of incoming terms, or emissions of terms), and data variable updates (denoted by classical assignments). Statements can also be built considering the following control primitives: sequence (;), condition (if-statements) or counted-repetition (for-statements).

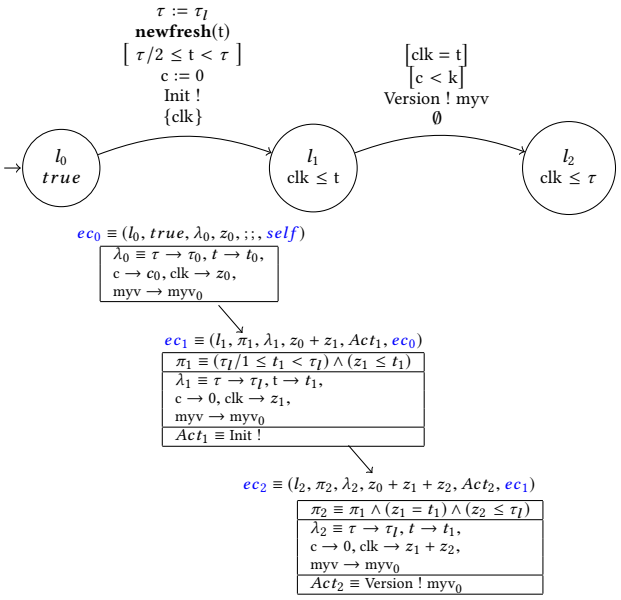

Figure 3: Illustration of the symbolic execution of a Symbolic Timed Automata (STA)

Figure 3 depicts a three-locations STA excerpt of the STA modeling a behavior of a Trickle (given in Figure 4): its locations are l_0 (the initial location), l_1 and l_2. Locations are

associated with invariants as in timed automata, yet in STA allow clocks to be bounded by first-order terms (being a data variable or another composite term): the invariant of location l_0 is $true$, the duration to remain in l_0 is unconstrained which defines different initialization instants of the STA (nodes are not synchronised), whereas, the invariant of location l_1 (resp. l_2) is clk $\leq$ t (resp. clk $\leq \tau$) where the data variable t (resp. τ) is used as a bound in the invariant.

Each transition is composed of a source state, a target state, a sequential statement, and set of clocks to be reset.

Transition $l_0 \rightarrow l_1$ is composed of the following statements: it resets the Trickle interval to the smallest value τ_l (assignment $\tau := \tau_l$); assigns t with a new fresh value (action newfresh(t)); constrains this new value of t to be within the second half of the τ-interval (guard $\tau/2 \leq$ t $< \tau$); resets the redundancy counter c (assignment c := 0); specifies the emission of an initialization signal to the environment on port Init (abstracted by the communication action Init!); and finally resets the clock clk which is used to constrain instants at which Trickle timer events occur.

Transition $l_1 \rightarrow l_2$ is composed of statements which encode the Trickle emission (communication action Version!myv) exactly at moment when the chosen new value for t elapses, which is measured since the clock clk has been reset by previous transition (guard clk = t). The transition is constrained as well by counter c not reaching its bound (guard c < k), i.e., the count of redundant versions received from neighbours is still less than k, the redundancy constant.

Symbolic execution. We suggest to use techniques of symbolic execution [10] to simulate timed behavior of STA. Symbolic execution consists in executing its transitions using symbolic parameters (rather than concrete values) which results in a set of reachable extended-locations that are denoted symbolically in the form of Execution Contexts (abbr. EC). An EC $ec \equiv (l, \pi, \lambda, \theta, Act, pec)$ is composed as follows:

(1) the reached location l that determines which transitions can be executed next,
(2) a formula π, the so-called Path Condition (abbr. PC), accumulating all temporal and data constraints induced by previously encountered guards and statements,
(3) a substitution λ associating terms over symbolic parameters to system variables,
(4) a sum of symbolic durations θ elapsed so-far since the beginning of the execution,
(5) a sequence of communication actions Act,
(6) an EC pec, called predecessor EC, giving access to the EC from which ec has been built.

Symbolic execution of the excerpt STA is illustrated in Figure 3. It allows reaching an EC ec_1, the time spent in this context is denoted by the symbolic duration z_1. The time elapsed so-far is $z_0 + z_1$ as the predecessor of ec_1 is ec_0, the time spent in this latter context is denoted by z_0. From the definition of ec_1, one can see as well that t is assigned with a new symbolic parameter ($\lambda_1(t) = t_1$, initially $\lambda_0(t) = t_0$) and that this value occurs in the second half of the current interval τ (see sub-formula ($\tau_l/2 \leq t_1 < \tau_l$) of PC π_1, with $\lambda_1(\tau) = \tau_l$). The other part ($z_1 \leq t_1$) of PC π_1 constrains the duration z_1 to remain in the location l_1 by t_1, which corresponds to the first half of the current interval, i.e., the listen-only period of the node at the end of which it transmits its version.

The symbolic execution of a transition is computed in three intermediate steps: i) statements, including guards, are evaluated, ii) and then the set of clocks to be reset are assigned with zero, iii) and finally since that, all clocks are advanced of the same amount of time (denoted by symbolic duration) which represents the time that will be spent in the target location, and naturally the invariant of the location is evaluated too.

The symbolic execution of $l_1 \rightarrow l_2$ from ec_1 results in an EC ec_2: it denotes that node transmits at the randomly chosen time instant denoted by t_1 (see sub-formula $z_1 = t_1$ of PC π_2). The emission is also symbolically denoted by Version!myv$_0$, the emitted symbolic parameter (myv$_0$) represent all possible values that can be assigned with myv at this step of execution. The single clock clk has been advanced of a symbolic duration since being reset by previous transition (λ_2(clk) = $z_1 + z_2$). We can see that on the difference of numeric simulation a single evaluation by symbolic execution captures many numeric executions, this because variables, including clocks are assigned with constrained symbolic parameters (by the inferred PC), rather than concrete ones.

Models in the Diversity tool can be atomic (here an STA) or compositional, with the possibility to define generic model template that can be instantiated many times. Figure 4 depicts a template STA modeling the generic behavior of a Trickle-node. Two excerpt transitions of the automaton have been already discussed in details, overall the automaton faithfully implements the rules of the Trickle algorithm (see Section 2). We use asynchronous communication mechanisms provided by the tool, to specify message-passing between nodes (over unbounded fifo-queues) where sending messages is not blocking for the sender node. This is actually the case when dealing with WSN kind of connectivity.

Scenarios are computed using the SE of the overall compositional model under interleaving unfolding. The latter consists in executing a transition at a time, of one of the component STA of the model and making a tuple of execution contexts (one per STA) evolve accordingly. This reached tuple of contexts, that we call a System Context (abbr. SC) allows us to characterize a scenario of messages exchanged so-far. Intuitively, the evolution of the overall execution will

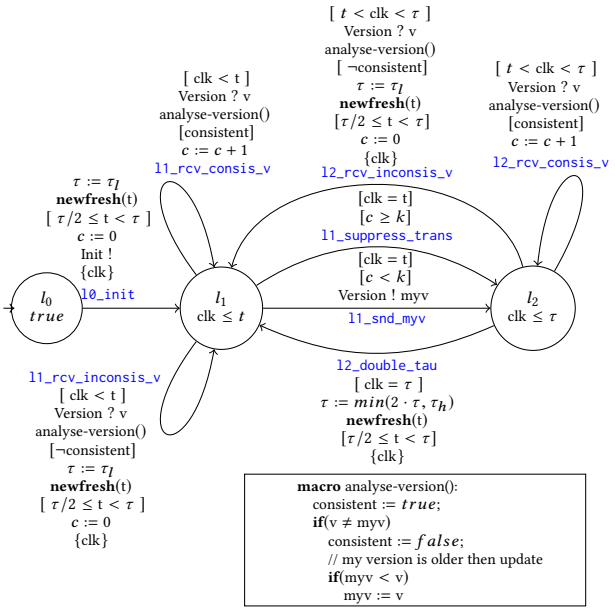

Figure 4: Model of a Trickle-node behavior.

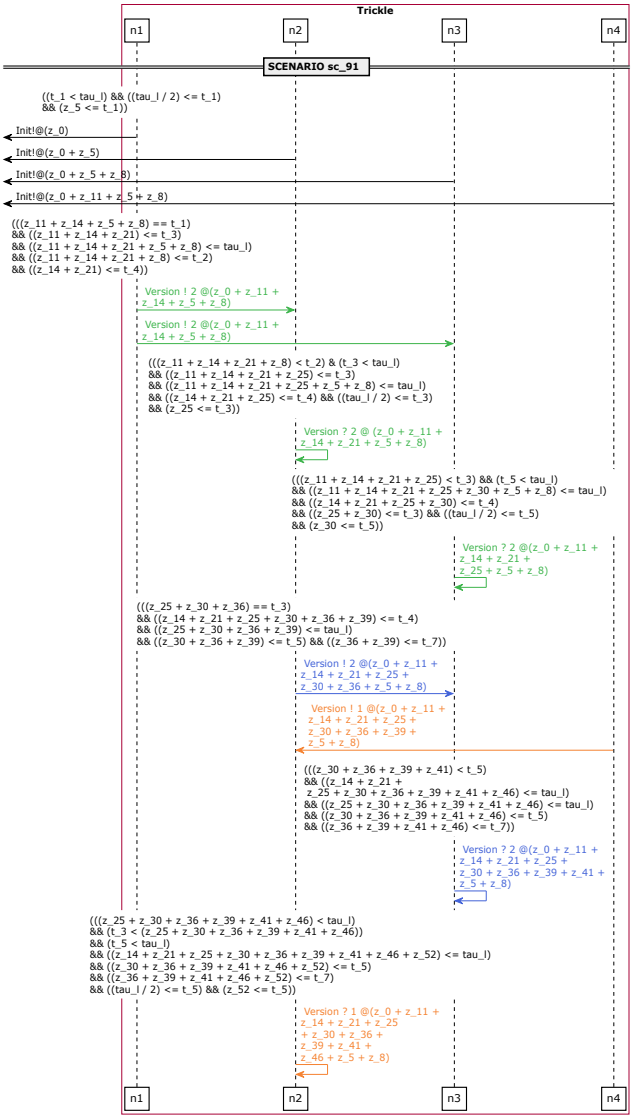

Figure 5: Outdated node situation - sequence diagram generated by our tooling.

essentially concern the EC relating to the STA whose transition is being executed and results in a new SC. We omit the illustration of an SC for readability sake, and we give the corresponding execution as a sequence diagram in Figure 5. The latter depicts an example of a scenario computed from a four-nodes model (Figure 1, with $i = 3$).

In the sequence diagram of Figure 5, pairwise send/receive events are of the same color. Every send/receive event is labeled by its timestamp as a sum of symbolic durations. We show as well parts of PC occurring on nodes to better how those timestamps in particular are constrained. For clarity, we have given versions numeric concrete values: n_1 is assumed to be the frontier node, it propagates a new version (2), other nodes n_2–n_4 hold an older one (1). The scenario shows that n_4 is outdated (k = 1): Its only neighbor who can send him versions, n_3, suppressed its transmissions, n_3 has been updated with the new version together with n_2 (by n_1, see the broadcast events in green), then its counter c is subsequently reset ($c = 0$); therefore when n_3 receives the same (new) version once again, from n_2, it increments its counter c ($c = 1$) that reaches k (see events in blue).

4 HEURISTIC-AIDED CONFIGURATION

Hit-Or-Jump heuristic. We ground our approach on the heuristic Hit-or-Jump (HoJ) [3] which is implemented by the Diversity tool [15]. The primary use of the heuristic is in Model-Based Testing (MBT) approaches: It allows the selection of tests for complex communicating systems modeled by product of automata, and hence are exposed to the well-known state space explosion. The heuristic is a generalisation of exhaustive search with random walks, and can be described as follows (see lower part of Figure 6):

- The targeted coverage is defined by a re-specified sequence of transitions (or states), possibly non-consecutive as it is difficult in general to guess a strict sequence when it comes to automata product.
- The heuristic conducts a mini-exploration, of the model which is limited in depth (local height), the latter is a

sensible parameter of the heuristic (in case of combinatorial explosion in width). During this mini-exploration, the heuristic observes the coverage progress. If the coverage is complete then the heuristic asserts, otherwise it randomly selects a number of reached SCs of maximal coverage (hit count) from which the next mini-exploration is repeated. The mini-explorations can be bounded (trials count), when all reached SCs do not cover anything, a some of them can be chosen at random to continue the exploration (jump count).

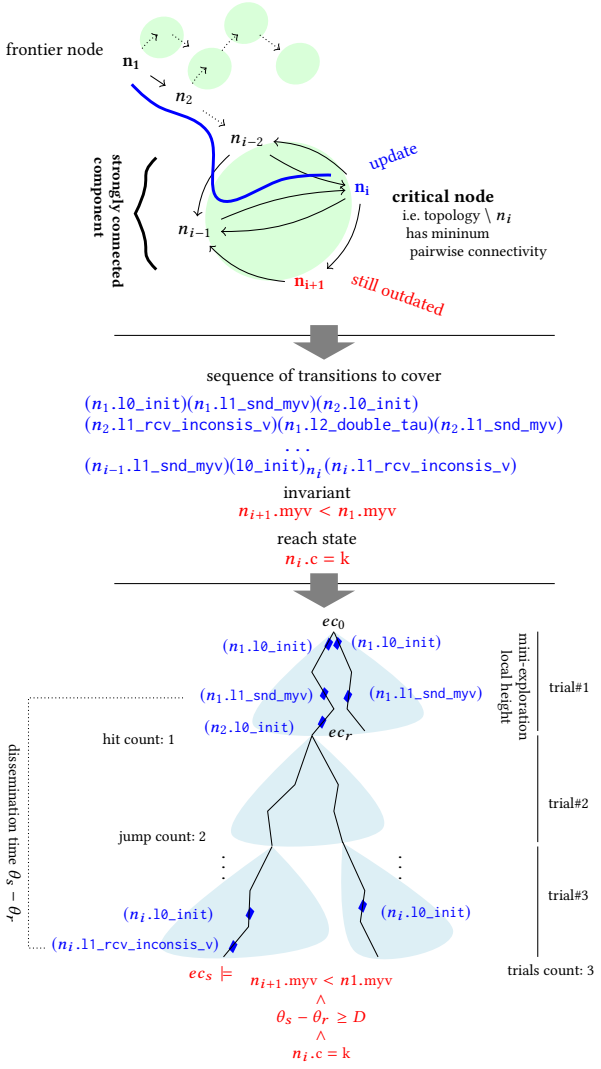

Figure 6: Heuristic search for outdated nodes.

Configuration aided by Hit-or-Jump. We use the heuristic on the STA models that we have developed in Section 3. The approach is guided by outdated nodes' situations search.

We start with first values for Trickle parameters, run the heuristic (a number of times), look for outdated nodes' situations after a certain period of time (denoted by D). Then we adjust those parameters and re-run the heuristic (again a number of times) for those new values, if the number of outdated nodes decreases or those are found with lower success rate of the heuristic, then we conclude that those parameters are likely to be more efficient than the previous ones. After some runs, if the number of outdated nodes does not decrease or decreases slightly, we stop the search.

As glimpsed before, the heuristic attempts to cover a prespecified sequence of transitions which characterizes the targeted behavior. Unlike in MBT where such sequences often encode a coverage criterion, in our approach those sequences are meant to highlight outdated nodes' situations. For this, we put in place a strategy to define those sequences based on the connectivity of the topology. In favorable cases, those sequences are sufficient to run our experiments and highlight a reasonable set of outdated nodes' situations. The strategy is illustrated in Figure 6: a set of *critical nodes*[2] is identified. They are critical since if removed, the residual topology graph will have minimum pairwise connectivity. Updating a critical node will eventually trigger the update of the nodes forming the involved (maximal) strongly connected component, an outdated node can be among those. The overall process is illustrated in Figure 6.

The sequence of Figure 6 denotes a chained update, starting from frontier node n_1 until reaching the critical node n_i assuming that nodes are initialized in a desynchronized manner. The propagation of a new version from an intermediate node n_j to its successor n_{j+1} (that is (n_j.l1_snd_myv), (n_{j+1}.l1_rcv_inconsis), then (n_{j+1}.l1_snd_myv)) requires time elapsing of at least $\tau_l/2$ in order for n_{j+1} to propagate on its turn the new version at t of n_{j+1} chosen in the second half of τ_l ($\tau_l/2 \leq t < \tau_l$, see Figure 4). As time elapses with big steps ($\tau_l/2$), some nodes will change their locations, and hence either send their versions (gossip) or double their current interval. Typically, in the previous case n_j will be in location l_2 (after its transmission) and inevitably has to double its current interval (that is (n_j.l2_double_tau)) in order to reach again location l_1.

Experimentation. We have created different networks composed of resp. 4, 8, 16 nodes, the behavior of each node is modeled by an STA of Figure 4. The objective is to investigate dissemination of updates within a given D period of time, by varying the value of the redundancy constant k. Using

[2]Recognition of a set of critical nodes (of given size) is a general graph problem, roughly speaking, upon the removal of one of those nodes, the strongly connected component to which it belongs, will be decomposed into several ones [18].

our tooling[3] (see screenshot Figure 7), we have profiled 100 heuristic runs for each D and k. Then, we have analyzed frequency of execution scenarios which highlight outdated nodes' situations in the neighborhood of the critical nodes, in the sens of Figure 6. We have observed that those are more frequent as k is low (i.e. less than 2), same for D. On a PC equipped with an Intel Core i7 processor (7^{th} generation) and 32GB RAM, we have found scenarios with outdated nodes in 30s (resp. 379s) average time, for $k = 2$ (resp $k = 3$) and $D = 3$ (resp. $D = 7.5$). The symbolic simulation is combinatorial, but our heuristic helps us to control to a certain extent the number of explored behaviors, the computation time is consumed in large majority by SMT solvers (CVC4 or Z3 or YICES) which are called to analyze the temporal constraints (see Figure 5).

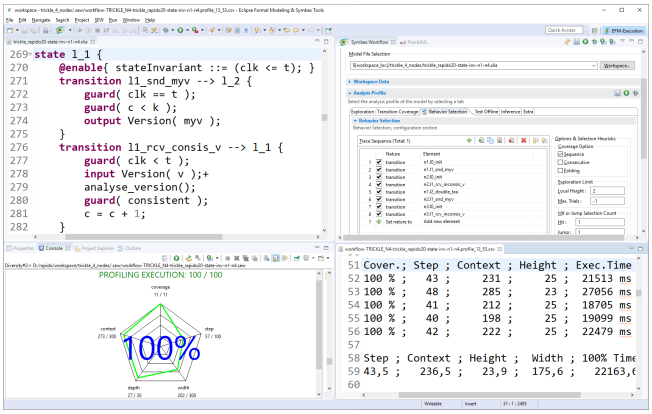

Figure 7: Tooling for editing STA model & profling Symbolic Simulation.

5 CONCLUSION

We have developed an original approach for configuring Trickle-based wireless sensors networks, the approach relies on a model-exploration heuristic guided by the network pairwise-connectivity. The heuristic runs on nodes' timed behavioral models, a bounded symbolic simulation involving some randomness, with the objective to evaluate the efficiency of Trickle configurations to quickly propagate new versions among nodes. The symbolic simulation allows for an efficient compact representation of the state space, on the other hand the heuristic makes the search practicable in the case of many-nodes network. First experiments show that the heuristic concludes often, reporting on the number of outdated nodes for each configuration. In the future, we plan experiments under more complex network for example with respect to critical nodes distribution, and also consider dynamic networks where the number of nodes is not constant.

[3]Tooling available on ftp://ftp.cea.fr/incoming/y2k01/rapido_20, implemented on top of the tool Diversity distributed open source by the project Eclipse Formal Modeling (EFM), https://projects.eclipse.org/projects/modeling.efm

REFERENCES

[1] Re-Emerging Mirai-like Botnets Are Threatening IoT Security in 2019. http://www.iotforall.com/mirai-botnets-threatening-iot-security-2019/.

[2] RPL: Ipv6 routing protocol for low-power and lossy networks, request for comments: 6550. Technical report, Cooper Power Systems and Cisco Systems and Stanford University, March 2012.

[3] R. Cavalli A., Lee D., Rinderknecht C., and Zaïdi F. Hit-or-jump: An algorithm for embedded testing with applications to IN services. In *FORTE*, 1999.

[4] R. Alur and D. Dill. A theory of timed automata. *Journal Theoretical Computer Science*, 1994.

[5] M. Becker, K. Kuladinithi, and C. Görg. Modelling and simulating the trickle algorithm. In *MONAMI*. Springer.

[6] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 1983.

[7] T. Coladon, M. Vucinic, and B. Tourancheau. Multiple redundancy constants with trickle. In *PIMRC*. IEEE, 2015.

[8] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *IEEE ICLCN*, 2004.

[9] J. Hui and R. Kelsey. Multicast protocol for low-power and lossy networks, request for comments: 7731. Technical report, Silicon Labs, February 2016.

[10] C.King J. Symbolic execution and program testing. *Communications of the ACM, Volume 19*, July 1976.

[11] H. R. Kermajani, C. Gomez, and M. H. Arshad. Modeling the message count of the trickle algorithm in a steady-state, static wireless sensor network. *IEEE Communications Letters*, 2012.

[12] P. Levis, T. Clausen, J. Hui, O. Gnawali, and J. Ko. The trickle algorithm, request for comments: 6206. Technical report, March 2011.

[13] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *TinyOS: An Operating System for Sensor Networks*. Springer, 2005.

[14] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Int. Symp. NSDI*. USENIX Association, 2004.

[15] Arnaud M., Bannour B., and Lapitre A. An illustrative use case of the DIVERSITY platform based on UML interaction scenarios. *Electr. Notes Theor. Comput. Sci.*, 2016.

[16] T. M. M. Meyfroyt. An analytic evaluation of the trickle algorithm: Towards efficient, fair, fast and reliable data dissemination. In *WoWMoM*. IEEE.

[17] N. M. T. Nguyen, B. Bannour, A. Lapitre, and P. Le Gall. Behavioral models and scenario selection for testing iot trickle-based lossy multicast networks. In *VVIoT@ICST*. IEEE, 2019.

[18] N. Paudel, L. Georgiadis, and G. F. Italiano. Computing critical nodes in directed graphs. *ACM Journal of Experimental Algorithmics*, 2018.

[19] C. Robinson-Mallett, R. M. Hierons, and P. Liggesmeyer. Achieving communication coverage in testing. *ACM Software Engineering Notes*, 2006.

[20] Von Styp S., C. Bohnenkamp H., and Schmaltz J. A conformance testing relation for symbolic timed automata. In *FORMATS*. Springer, 2010.

[21] M. Vucinic, M. Król, B. Jonglez, T. Coladon, and B. Tourancheau. Trickle-D: High fairness and low transmission load with dynamic redundancy. *IEEE IoT Journal*, 2017.

System Simulation with PULP Virtual Platform and SystemC

Éder F. Zulian, Germain Haugou, Christian Weis, Matthias Jung and Norbert Wehn

System Simulation with PULP Virtual Platform and SystemC

Éder F. Zulian
TU Kaiserslautern
Germany
zulian@eit.uni-kl.de

Germain Haugou
ETH Zürich
Switzerland
haugoug@iis.ee.ethz.ch

Christian Weis
TU Kaiserslautern
Germany
weis@eit.uni-kl.de

Matthias Jung
Fraunhofer IESE
Kaiserslautern, Germany
matthias.jung@iese.fraunhofer.de

Norbert Wehn
TU Kaiserslautern
Germany
wehn@eit.uni-kl.de

ABSTRACT

Driven by performance, power and energy requirements compute platforms evolved from single-core homogeneous into highly parallel heterogeneous architectures. These architectures use different CPUs, GPUs, accelerators, interconnects, memories, etc., and target diverse applications creating a vast design space. Moreover, ever-growing security concerns leverage the adoption of open source silicon designs and tools, specially those with collaborative research and engineering efforts from industry and academia to develop and maintain for the long term. In this context, RISC-V based solutions are an outstanding choice and virtual platforms are essential for a fast design cycle. However, simulation frameworks often do not provide off-the-shelf interoperability hindering the reusability of existing models.

In this paper, we present for the first time a coupling of the PULP virtual platform, which provides manycore RISC-V accelerators in a feature-rich simulation environment, with SystemC/TLM-2.0, a de facto standard in industry that is also largely used in the academia. Furthermore, we evaluate the coupling of simulation engines and demonstrate its usefulness in a case study.

CCS CONCEPTS

• **Computing methodologies** → **Discrete-event simulation**; **Simulation environments**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**.

KEYWORDS

RISC-V, PULP, Virtual Prototyping, Heterogeneous Systems, System-level Modeling, SystemC/TLM

ACM Reference Format:
Éder F. Zulian, Germain Haugou, Christian Weis, Matthias Jung, and Norbert Wehn. 2020. System Simulation with PULP Virtual Platform and SystemC. In *Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '20), January 21, 2020, Bologna, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3375246.3375256

RAPIDO '20, January 21, 2020, Bologna, Italy

ACM ISBN 978-1-4503-7777-5/20/01…$15.00
https://doi.org/10.1145/3375246.3375256

1 INTRODUCTION

Recently, Google and other founding partners from industry and academia announced OpenTitan [10] the first open source silicon root of trust (RoT) project. According to its announcement, OpenTitan will deliver a high-quality RoT design and integration guidelines for use in data center servers, storage, peripherals, and more. OpenTitan is based on RISC-V, hence it provides us with a prominent example for the huge impact of the RISC-V ecosystem on industry. Moreover, GAP8 [7] is another example from industry that is a derivative from PULP [26] that originated in the academia. The System-on-Chip (SoC) designed by GreenWaves targets edge computing and Internet of Things (IoT) systems consisting of autonomous energy-constrained devices. The SoC relies on the PULP RISC-V architecture, it has nine RISC-V cores, I/O peripherals and a hardware convolution computation engine specialized for use with convolutional neural networks (CNNs) related applications.

Today's huge diversity of applications, system heterogeneity, performance and energy-efficiency requirements, along with a vast number design choices, expose system-level architects to an ever-growing complexity when developing compute platforms. In this context, *Virtual Platforms* (VPs) are not only a valuable resource, but also mandatory to: (1) Facilitate rapid research on all software and hardware layers. (2) Manage the inherent complexity of contemporary system's *Design Space Exploration* (DSE). (3) Accelerate the design cycle, anticipate analysis, optimization, verification. Moreover, SystemC and TLM are largely used in academic research and it is a de facto industry standard, which is successfully used by leading companies to advance their products by means of VPs.

The purpose of this work, therefore, is to present the coupling of *GVSoC*, the simulation engine of the PULP VP, with SystemC/TLM. For the first time, researchers and industry can profit from models provided by the PULP VP, such as manycore RISC-V accelerators and several peripherals, in a SystemC simulation environment. The coupling allows PULP VP components to be combined with SystemC compatible models to form heterogeneous simulation frameworks. Moreover, the source code developed for the coupling along with artifacts, such as SystemC modules and scripts for automated setup of prerequisites, are open source and committed to official repositories of the PULP project. Furthermore, we believe that the

interoperability of simulation engines not only increases their application scope, but also facilitates further extensions while promoting their use and long-term maintenance.

The rest of this paper is structured as follows: Section 2 discusses related work. An overview of the PULP VP focused in its simulation engine *GVSoC* along with SystemC/TLM-2.0 is provided in Section 3. The coupling is explained in Section 4. Further, in Section 5 we evaluate essential aspects of our work while demonstrating its usefulness with case studies that can be used as entry point for the creation of heterogeneous simulation frameworks. Section 6 concludes the paper.

2 RELATED WORK

A vast body of research has proposed a multitude of approaches and resourceful solutions to create VPs. As expected, the trade-off accuracy vs. simulation speed is a constant concern in virtually all works. Moreover, considerable part of these works focus not only on breakthrough innovations started from scratch, but also on the integration, extension and reuse of existing solutions. We summarize some recent achievements in the following.

The authors of [4] present an integrated virtual prototyping solution called VPSim that promises exceptional performance, easy of use and automation of modelling and exploration. The proposed VP uses QEMU [1] linked to SystemC TLM-2.0 modules. Performance is granted by using the TLM-2.0 specialized transport interface *DMI* at the cost of accuracy, but the solution allows different accuracy levels for specific regions of interest (ROIs). Similar approach is presented by [15], where a processor model of a recent ARMv8 ISA is generated with a QEMU-based CPU emulator framework called Unicorn [20]. The authors use SystemC to integrate the processor model into a VP in combination with a few third party SystemC/TLM peripherals from [27]. Temporal decoupling is used to increase simulation performance.

Ptolemy [5] and *FERAL* [17] allow multiple models of computation (MoCs) in a hierarchical heterogeneous design environment. Moreover, *FERAL* allows the integration of compiled/linkable tools for which the source code is available with closed-source or non-compiled frameworks. Different MoCs are also integrated in [21], where a new approach for virtual prototyping of analog and mixed-signal embedded (AMS) systems based on SysML is proposed. The authors extract low-level cycle accurate prototypes from the high-level abstraction models and present the integration of analog components into a virtual prototyping framework. Another characteristic of *FERAL*, also used by the *Structural Simulation Toolkit* (SST) [22], is the use an intermediate software layer to bind simulation components into an integrated environment. According to its documentation, the SST was designed to explore highly concurrent systems where the ISA, microarchitecture, and memory interact with the programming model and communications system. Recently, a parallel simulation environment based on MPI was added to the toolkit, what may counterbalance any overhead generated by the middleware layer. Parallelization is a relevant option for accelerating simulations that is also explored in [28] that uses temporal decoupling and a parallel SystemC kernel called SCope [29] to speedup simulations of a SystemC-based MPSoC platform formed by a SystemC-wrapped Instruction Set Simulator (ISS) together with peripherals. Moreover, further speedup is achieved by skipping the simulation of processors in idle state.

An extendable translating instruction set simulator is presented in [19]. The proposed framework does binary translation and implements a plugin mechanism that allows to quickly include new functionality whenever an interrupt is received, e.g., into the translation stage, the simulation loop, during accesses to the memory. Moreover, the framework has special focus on virtual prototypes written in SystemC/TLM and its plugins include tracing tools, SystemC interfaces, peripherals and fault injection capabilities.

The authors of [25] create a heterogeneous VP which consists of a SystemC-based GPU attached to the existing open source SystemC-based framework SoCRocket [6]. Besides GPU, the proposed VP supports the open source LEON CPU and GRLIB components. According to the authors RTL implementations are provided what makes possible a feedback loop between simulation results into realistic hardware models. Similarly, the PULP project provides open source RTL implementations. In [16] a memory simulator called NVMain is extended with a reference implementation of a new SRAM cache and implementation of various latency optimizations for die-stacked cache. In this work, SystemC/TLM models from DRAMSys [14] build a memory subsystem used in a case study for evaluation purposes.

The authors of [12] proposed and implemented a RISC-V based virtual prototype in SystemC/TLM that provides detailed models of a RISC-V core, an interrupt controller and essential peripherals that can be used for system-level exploration. RISC5, an implementation of a RISC-V ISA in gem5, was presented in [23]. According to its authors, RISC5 is validated against performance data from the Chisel C++ emulator and FPGA soft core and is shown to have less than 10% error on several performance statistics. Moreover, [23] was further evaluated and extended by [24]. Also in gem5, the authors of [18] provided support to SystemC co-simulation. This was a key step that allowed the reuse of openly available models developed for more than a decade in joint effort of researchers and industry and simulation workloads together with SystemC-based models.

3 BACKGROUND

This section provides a brief overview of the PULP project along with the PULP VP and SystemC/TLM-2.0 with focus on characteristics that are relevant to this work.

3.1 PULP Project

PULP [26] is an open source parallel architecture. One of its configurations has a main core and a main memory called *Accelerator DRAM* connected to a cluster of RISC-V cores sharing a local memory and using a DMA engine to transfer data between the accelerator DRAM and the cluster internal memory. Tasks can be offloaded to the PULP accelerator cluster. The RISC-V core(s) within the accelerator that are assigned to the task uses a DMA engine to transfer data between memories.

3.2 PULP Virtual Platform

The PULP Virtual Platform is part of the PULP project, hence it evolves and is maintained along with the PULP ecosystem. The

simulation engine of the PULP VP is called *GVSoC*, which is developed in Python and C++ using a component-based approach. Each component has client and server interfaces. Similarly to gem5 [2], the instantiation and binding of all components is done in Python while the models are written in C++. Python is used to control the simulation but once it is running, the execution uses only C++. The whole platform is a set of parametric models. Each target is described by a JSON description of the architecture which is driving the instantiation, connection and configuration of each component.

The execution model is using an event-driven approach. Any activity in a model is simulated as a function call which executes until completion. To correctly model frequency scaling, each model is part of a clock domain and the model is scheduling its activity using cycles. An activity can be registered in the future by specifying a number of cycles so that the model is not affected by a frequency change. A *Top Time Engine* is responsible for scheduling the various clock domains according to their activity. Time stubs are automatically inserted when two components of two different clock domains are bound together to apply timing conversions. Performance is modeled by scheduling activities at a specific time. For example, the function modeling an instruction will enqueue another activity two cycles later to model the fact that the instruction took two cycles to execute. Accuracy depends on properly timing the core pipeline (data dependency, branch penalty and so on). The interconnects are also timed so that they can properly model bandwidth and latency, using an asynchronous protocol. A master can send multiple outstanding requests and the slave can grant them and reply to them asynchronously so that advanced behaviors can be modeled. This approach resembles the TLM-2.0 approximately-timed (AT) coding style with non-blocking transport interface.

3.3 SystemC & TLM

SystemC is a C++ class library and an industry standard defined in the IEEE Standard for Standard SystemC® Language Reference Manual [13] often used in combination with TLM. A robust and mature proof-of-concept simulator is made openly available by Accellera Systems Initiative. TLM has two main focuses: transaction-based *communication* and *interoperability* of modules. A TLM transaction comprises a collection of pin wiggles, hence it allows a substantial reduction in the number of events that have to be handled by the event-driven simulation kernel reducing the simulation time. Interoperability can be achieved for non-native SystemC/TLM models with the use of *wrappers*, *transactors*, *adapters* and *bridge components*.

4 COUPLING PULP VP AND SYSTEMC

In this section we explain the most relevant aspects of the coupling of the PULP VP and SystemC/TLM-2.0.

An overview of the coupling is shown in Figure 1. The required changes and new open source components created to this end are detailed in the following.

4.1 Preliminary Aspects

The coupling is achieved by executing the *Top Time Engine* of *GVSoC* (the simulation engine of the PULP VP) as a SystemC thread as shown in Listing 1, hence it is scheduled for execution by the

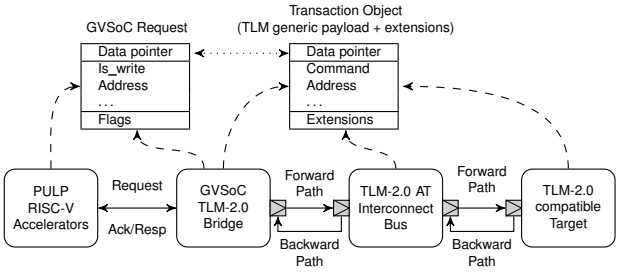

Figure 1: Transaction objects are constructed on-demand in the *Bridge*. The data pointer within a request is referenced in the transaction avoiding the copy/movement of data.

Listing 1: *Top Time Engine* within a SystemC thread

```
SC_MODULE(gvsc_module) {
  SC_HAS_PROCESS(gvsc_module);
  gvsc_module(sc_module_name n, vp::time_engine *eng) :
    sc_module(n), engine(eng) {
    SC_THREAD(run);
  }
  void run() {
    eng->run_loop();
  }
  vp::time_engine *engine;
};
```

SystemC simulation kernel. Moreover, at the beginning of a simulation, during the transition from the Python configuration phase to the C++ execution phase an initialization function is called by the thread running the *Top Time Engine* to execute the elaboration phase in which all *GVSoC* and SystemC modules are instantiated. Subsequently, `sc_start()` is called.

Besides the *Top Time Engine*, other two components are used to establish communication and integration of *GVSoC* and SystemC components. They are: (1) The *GVSoC TLM-2.0 Bridge*, a transactor responsible for the translation of *GVSoC* requests into TLM transactions and vice versa. (2) The *Approximately-Timed (AT) Interconnect Bus*, a memory mapped interconnect bus where any number of TLM compatible components can be attached.

4.2 GVSoC Top Time Engine

To have a close interaction between *GVSoC* and SystemC schedulers, *GVSoC Top Time Engine* is embedded in a SystemC thread as shown in Listing 1. Every time this special thread is scheduled for execution by the SystemC kernel, it executes all the activities scheduled for the current simulation time, allowing modules from both sides to create new events. Once no more activity has to be executed, it waits until the simulation time is advanced by the SystemC simulation kernel to the time of the next activity. It is possible that an external event coming from the SystemC domain, for example, an external access, creates an activity before the next scheduled one, in which case it wakes up the *GVSoC* thread for proper treatment of the external event. As shown in Listing 2, the *Top Time Engine* of *GVSoC* waits for either an amount of time when the next activity is known and internally scheduled or an activity that may happen before that originated in the SystemC domain. It is also possible that *GVSoC* scheduler has nothing to schedule and is just waiting for an event (using the same mechanism) which is used to wake it up to execute the new activity.

Listing 2: Main changes in the GVSoC scheduler

```
while(1) {
  if (!first_client) {
    if (stop_req || locked)
      break;
    // Wait for events from the SystemC domain
    wait(sync_event);
  } else {
    int64_t now = (int64_t)sc_time_stamp().to_double();
    int64_t ne_time = first_client->next_event_time;
    vp_assert(ne_time >= now, NULL, NULL);
    // Wait next event or an asynch event from SystemC
    wait(ne_time - now, SC_PS, sync_event);
    now = (int64_t)sc_time_stamp().to_double();
    if (now == first_client->next_event_time)
      break;
  }
}
```

4.3 GVSoC TLM-2.0 Bridge

The *GVSoC TLM-2.0 Bridge* consists of a transactor component that converts *GVSoC* requests into TLM transactions and vice versa. Hence, it allows native *GVSoC* models to communicate with any TLM-compliant component. The bridge is TLM-2.0 base protocol compliant, hence it respects the exclusion rules defined by the protocol. The approximately-timed coding style is adopted, therefore transactions are initiated in the forward path using the non-blocking transport interface. Moreover, it honors timing annotations and supports responses via both backward and return paths, including early completion and implicit end request (*END_REQ*).

For the sake of illustration, in a typical scenario a memory access request from the DMA engine is a few kilobytes in size, then it relies in a DRAM controller to perform multiple memory requests. In each request 64 bytes of data are exchanged between DRAM controller and the DRAM through a 64-bit wide data bus in a burst consisting of 8 beats, each beat 8 bytes. The *GVSoC TLM-2.0 Bridge* breaks a memory access request into multiple parts adequate to its immediate target. For that, it allocates new TLM transaction objects on-demand using a *memory manager*, which contains a pool of transaction objects for reuse. In the context of TLM, a memory manager is a common/mandatory C++ class/object used to avoid recurrent construction and deletion of transaction objects. Moreover, the bridge uses TLM generic payload extensions to check the integrity of transactions and the request they derive during the communication process between PULP components modeled by *GVSoC* and SystemC modules. Ignorable extensions are used as they are transparent to other components allowing transactions to be normally routed through any number of interconnect components in their way to the target.

Table 1 shows the relevant fields of the *GVSoC* request along with their equivalents in the TLM transaction. The `is_write` flag of the *GVSoC* request is used to define the TLM command, read or write, the `size` defines the number of parts a *GVSoC* request is split into. Each of the parts is a TLM transaction. The `data_length` of TLM transaction is a configuration that must fit requirements from the target side. In order to avoid movement of large amounts of data, the same data region pointed by the `data pointer` in a *GVSoC* request is directly referenced by TLM transactions. The pointer to the base of the data region is incremented with an offset relative to the part of the *GVSoC* request a transaction corresponds to. Similarly, the `address` field is reused and incremented by an offset. `Flags` are GVSoC-specific, therefore they are not propagated. Other fields of the TLM transaction such as `byte_enable` and `streaming_width` are initialized to their default values and `dmi_allowed` is not used in this work since the memory model design is approximately-timed, therefore a higher accuracy level that is more suitable for architectural exploration.

Table 1: GVSoC requests and TLM transactions

GVSoC Request	TLM Transaction	Observations
is_write flag	-	Request type
size	*data_length*	1:N
-	*command*	Defined by *is_write*
address	*address*	1:N (base + offset)
data pointer	*data pointer*	1:N (pointer + offset)
flags	-	-
-	*byte_enable*	-
-	*streaming_width*	-
-	*dmi_allowed*	-
-	*extensions pointer*	Integritycheck and routing

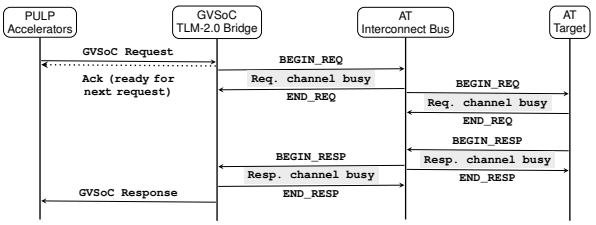

Figure 2: The *Bridge* and *Interconnect Bus* allow multiple transactions in-flight and flow control in both directions.

4.4 TLM-2.0 AT Interconnect Bus

The *TLM-2.0 Approximately-Timed Interconnect Bus* models a SoC interconnect memory mapped bus. The bus is the main junction point for the *GVSoC Bridge* and other TLM compatible modules/frameworks. It allows multiple connections in both upstream and downstream directions with *multi_passthrough sockets*, hence the number of initiators and targets is defined dynamically based on socket bindings. Moreover, it is designed to support address decoding functionality, but in its current state of development there is "*dummy*" decoding that serves as placeholder to be extended on-demand.

Figure 2 provides an illustration on how requests and responses are propagated. The four-phases of the TLM non-blocking protocol allows implementation of flow control in both directions.

5 EVALUATION

In the following we present experiments used to evaluate the functionality of the coupling. Different SystemC/TLM compatible components and simulators are used together to form a heterogeneous simulation framework/test scenarios in which proper scheduling and treatment of events is mandatory. Moreover, we use the open source *TLM-2.0 Base Protocol Compliance Checker* to analyse one million transactions that pass through our new components.

5.1 Heterogeneous Modeling Framework

As part of our evaluation we build the heterogeneous modeling framework shown in Figure 3. The framework consists in a SystemC/TLM simulation environment where RISC-V accelerators from

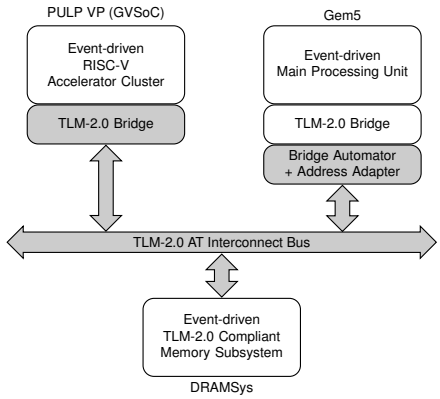

Figure 3: Example of a heterogeneous modeling framework. New components are highlighted in gray.

PULP (enabled by this work) are combined with other simulators, namely DRAMSys [14] and gem5 [2]. An overview of the simulators used along with open source components created to support the coupling is presented in the following.

DRAMSys [14] is a flexible memory subsystem design space exploration framework that consists of models reflecting the DRAM functionality, power consumption with a bind to DRAMPower [3], temperature behaviour and modeling of DRAM cell data retention. The framework provides models to build a complete memory subsystem, such as arbitration module, elaborate memory controller and multiple DRAM models. DRAMSys is particularly interesting for it is an AT TLM-compliant framework, hence it can be used for memory architectural DSE.

The gem5 simulator [2] is a modular platform for architecture research, encompassing system-level architecture as well as processor microarchitecture [9]. It provides models of diverse system components including CPUs, buses, caches, DRAM controllers [11] along with peripheral devices. RISC-V, x86, ARM, ALPHA and MIPS are among the ISAs supported [8]. Morevoer we profit from the accumulated achievements of [18] to couple gem5 to our environment and utilities provided by [30] facilitate the setup.

We create the *Gem5 TLM Bridge Automator* and an *Address Adapter*. The former parses gem5 configuration files to proper instantiate and initialize the key components provided by [18] such as the *Gem5SimControl* object. Moreover, it instantiates the *Address Adapter* component that can be used to convert memory addresses originated in gem5 to the address range of the *Interconnect Bus* and vice versa. Furthermore, we also provide a *Memory Manager* along with an example TLM-2.0 target component.

To evaluate the coupling we execute different workloads in the simulation framework shown in Figure 3. A RISC-V core within the PULP accelerator cluster executes a memory access pattern first writing values to the external DRAM (end point of the memory subsystem) and later reading them back from the DRAM. At the same time, gem5 executes Linux in a quad-core ARM based system that uses the aforementioned DRAM as main memory. In this case study the system simulated in gem5 and the RISC-V in the PULP VP use separate memory address ranges, hence different areas of the

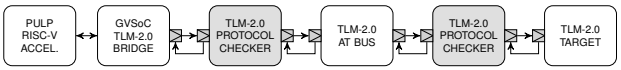

Figure 4: *TLM-2.0 Base Protocol Compliance Checker* usage.

Listing 3: Instantiation, binding and config (10^6 checks).

```
void scdomain::elab() {
  bridge = new gvsoc_tlm_br("br", this, ACCEPT_DELAY_PS, BPA);
  atbus = new ems::atbus("bus");
  pcbb = new tlm_utils::tlm2_base_protocol_checker<>("pcbb");
  pcbt = new tlm_utils::tlm2_base_protocol_checker<>("pcbt");
  pcbb->set_num_checks(1000000);
  pcbt->set_num_checks(1000000);
  bridge->isocket.bind(pcbb->target_socket);
  pcbb->initiator_socket.bind(atbus->tsocket);
  atbus->isocket.bind(pcbt->target_socket);
#ifdef __VP_USE_SYSTEMC_GEM5
  std::string cfg = get_config_str("tlm/gem5-config");
  g5automator = new ems::gem5_automator("g5automator", cfg);
  for (auto adpt : g5auto->adapters)
      adpt->isocket.bind(atbus->tsocket);
#endif /* __VP_USE_SYSTEMC_GEM5 */
#ifdef __VP_USE_SYSTEMC_DRAMSYS
  std::string sim = get_config_str("tlm/dramsys-config");
  dramsys = new DRAMSys("DRAMSys", sim);
  pcbt->initiator_socket.bind(dramsys->tSocket);
#else
  tgt = new ems::at_target("t", ACCEPT_DELAY_PS, LATENCY_PS, BPA);
  pcbt->initiator_socket.bind(tgt->tsocket);
#endif /* __VP_USE_SYSTEMC_DRAMSYS */
}
```

DRAM. Interactions with the memory subsystem consist of TLM transactions.

Requests from the RISC-V core in the PULP VP domain are translated into TLM transactions by the *GVSoC TLM-2.0 Bridge* and forwarded to the *TLM-2.0 AT Interconnect Bus*. TLM transactions coming from gem5, pass through our *Gem5 TLM Bridge Automator*, which instantiates the control module provided by gem5, and through our *Address Adapter* before being forwarded to the *TLM-2.0 AT Interconnect Bus*. Next, transactions from both domains go through the *TLM-2.0 AT Interconnect Bus* to the memory controller and to the DRAM, the last two implemented in DRAMSys.

Moreover, the third party simulators (gem5 and DRAMSys) are compiled as libraries and linked with the PULP VP platform to form a simulation environment on top of the SytemC simulation kernel. The environment is used, not only to test the scheduling capabilities of the modified simulation engine, but also as realistic stimuli generator for testing communication.

As result we observe millions of transactions generated in both domains properly routed through our new components to reach their targets. Further, the Linux system in gem5 is responsive, the values written to and read back from the memory by the RISC-V core are compared and match.

5.2 TLM-2.0 Protocol Compliance Check

We adapt the case study described in 5.1 to use the open source *TLM-2.0 Base Protocol Compliance Checker*, created by John Aynsley from Doulos, to analyse at least 10^6 transactions. Figure 4 illustrates how multiple instances of the protocol checker module are put in-line between the hops. Instantiation, binding and base configuration of relevant components are shown in Listing 3.

As result of our experiment, no protocol violations were observed.

Table 2: Execution Time (30 reps., Intel i7-4790 host CPU)

Memory Model	Maximum	Minimum	Average
PULP-VP Native	12.563 s	11.907 s	12.168 s
SystemC/TLM-2.0 AT[a]	17.792 s	16.621 s	17.071 s

[a]Implies use of SystemC-based scheduling.

5.3 SystemC Simulation Overhead

To estimate the runtime overhead introduced by the coupling, which results in the execution of the PULP VP on top of the SystemC simulation kernel, we create equivalent simulation scenarios with and without SystemC that we summarize in the following[1].

A workload, which consists of a simple compute kernel for matrix addition, manipulates data from a memory component accessed via DMA engine. The workload iterates 2000 times over a 100×100 matrix executing approximately 200 million instructions. Moreover, 8 RISC-V cores are used, hence there is also inter-core synchronization as the DMA is managed by one core, and each core needs to fetch a *chunk of data* to process (the DMA ensures coarse-grained accesses to the memory). The RISC-V cores within the cluster work on local scratchpad memory (128 KB) organized in 8 banks with a logarithmic interconnect with an average contention ratio of 10%. The matrices operated are stored in a memory component within the SoC, but outside the cluster, so there is a double buffering between the two memories for both input and output using the DMA.

Two versions of the memory component are created: (1) Native PULP-VP memory module. (2) SystemC/TLM-2.0 target memory, that is a simple implementation but in AT coding-style, non-blocking transport, four-phases protocol.

From our experimental results presented in Table 2 we observe about 40% increase in the simulation time when using our coupling.

6 CONCLUSION

The PULP architecture is resourceful with high relevance in industry and academia. The PULP Virtual Platform provides sophisticated models of the RISC-V accelerators and peripherals that accumulate considerable research/engineering efforts, hence the reuse of such models is of substantial importance.

In this work we presented, for the first time, a coupling that enables co-simulation of RISC-V accelerators cluster from the PULP VP in a SystemC environment. To reduce the engineering efforts in further couplings, all components created are open source and available for download. Furthermore, we evaluated the coupling and provided a case study that can be used as entry point for the creation of further heterogeneous simulation frameworks.

As future work we aim at reducing the simulation time overhead introduced by our coupling.

ACKNOWLEDGMENTS

This work was partially funded by the EU OPRECOMP project under grant agreement No. 732631 (http://www.oprecomp.eu) and also supported by the *Fraunhofer High Performance Center* for *Simulation- and Software-based Innovation*. We acknowledge the support of *Synopsys Inc.*.

[1]The interested reader may refer to [26] for further information on the PULP architecture.

REFERENCES

[1] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41. http://dl.acm.org/citation.cfm?id=1247360.1247401

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[3] Karthik Chandrasekar, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens. 2011. DRAMPower: Open-source DRAM Power & Energy Estimation Tool. http://www.drampower.info.

[4] Amir Charif, Gabriel Busnot, Rania Mameesh, Tanguy Sassolas, and Nicolas Ventroux. 2019. Fast Virtual Prototyping for Embedded Computing Systems Design and Exploration. In *Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '19)*. ACM, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/3300189.3300192

[5] Johan Eker, Jorn W Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, Yuhong Xiong, and Stephen Neuendorffer. 2003. Taming heterogeneity - the Ptolemy approach. *Proc. IEEE* 91, 1 (2003), 127–144. http://chess.eecs.berkeley.edu/pubs/488.html

[6] Luca Fossati, Thomas Schuster, Rolf Meyer, and Mladen Berekovic. 2013. Socrocket: a virtual platform for soc design. *DAta System In Aerospace (DASIA)* (2013).

[7] GeeenWaves. 2018. GAP8. https://en.wikichip.org/wiki/greenwaves/gap8. Last accessed 10 Nov 2019.

[8] gem5.org. 2011. gem5: Supported Architectures. www.gem5.org/Supported_Architectures. Last accessed 07 Nov 2019.

[9] gem5.org. 2011. The gem5 Simulator. http://gem5.org/Main_Page. Last accessed 07 Nov 2019.

[10] Google. 2019. OpenTitan. https://opensource.googleblog.com/2019/11/opentitan-open-sourcing-transparent.html. Last accessed 10 Nov 2019.

[11] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A.N. Udipi. 2014. Simulating DRAM controllers for future system architecture exploration. In *ISPASS 2014 - IEEE International Symposium on Performance Analysis of Systems and Software*. 201–210. https://doi.org/10.1109/ISPASS.2014.6844484

[12] V. Herdt, D. Große, H. M. Le, and R. Drechsler. 2018. Extensible and Configurable RISC-V Based Virtual Prototype. In *2018 Forum on Specification Design Languages (FDL)*. 5–16. https://doi.org/10.1109/FDL.2018.8524047

[13] IEEE. 2012. IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* (Jan 2012). https://doi.org/10.1109/IEEESTD.2012.6134619

[14] Matthias Jung, Christian Weis, and Norbert Wehn. 2015. DRAMSys: A Flexible DRAM Subsystem Design Space Exploration Framework. *IPSJ Transactions on System LSI Design Methodology* 8 (2015), 63–74. https://doi.org/10.2197/ipsjtsldm.8.63

[15] Lukas Jünger, Jan Henrik Weinstock, Rainer Leupers, and Gerd Ascheid. 2019. Fast SystemC Processor Models with Unicorn. In *Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '19)*. ACM, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3300189.3300191

[16] Asif Ali Khan, Fazal Hameed, and Jeronimo Castrillon. 2018. NVMain Extension for Multi-Level Cache Systems. In *Proceedings of the Rapido'18 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '18)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/3180665.3180672

[17] T. Kuhn, T. Forster, T. Braun, and R. Gotzhein. 2013. FERAL — Framework for simulator coupling on requirements and architecture level. In *2013 Eleventh ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2013)*. 11–22.

[18] C. Menard, J. Castrillon, M. Jung, and N. Wehn. 2017. System simulation with gem5 and SystemC: The keystone for full interoperability. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 62–69. https://doi.org/10.1109/SAMOS.2017.8344612

[19] Daniel Mueller-Gritschneder, Keerthikumara Devarajegowda, Martin Dittrich, Wolfgang Ecker, Marc Greim, and Ulf Schlichtmann. 2017. The Extendable Translating Instruction Set Simulator (ETISS) Interlinked with an MDA Framework for Fast RISC Prototyping. In *Proceedings of the 28th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype (RSP '17)*. ACM, New York, NY, USA, 79–84. https://doi.org/10.1145/3130265.3138858

[20] A. Q. Nguyen and H. V. Dang. 2015. Unicorn: Next Generation CPU Emulator Framework. https://www.unicorn-engine.org/. Last accessed 07 Nov 2019.

[21] Rodrigo Cortés Porto, Daniela Genius, and Ludovic Apvrille. 2019. Modeling and Virtual Prototyping for Embedded Systems on Mixed-Signal Multicores. In *Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '19)*. ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/3300189.3300193

[22] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. 2011. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.* 38, 4 (March 2011), 37–42. https://doi.org/10.1145/1964218.1964225

[23] Alec Roelke and Mircea R Stan. 2017. Risc5: Implementing the RISC-V ISA in gem5. In *First Workshop on Computer Architecture Research with RISC-V (CARRV)*.

[24] Robert Scheffel. 2018. *Simulation of RISC-V based Systems in gem5.* Master's thesis. Technische Universität Dresden.

[25] Patrick Siegl, Rainer Buchty, and Mladen Berekovic. 2016. Towards Bridging the Gap Between Academic and Industrial Heterogeneous System Architecture Design Space Exploration. In *Proceedings of the 2016 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '16)*. ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/2852339.2852343

[26] PULP Team. 2013. PULP Platform: Open hardware, the way it should be! https://www.pulp-platform.org/. Last accessed 10 Nov 2019.

[27] Jan Henrik Weinstock. 2018. Virtual Components Modeling Library. https://github.com/janweinstock/vcml. Last accessed 07 Nov 2019.

[28] Jan Henrik Weinstock, Rainer Leupers, and Gerd Ascheid. 2017. Accelerating MPSoC Simulation Using Parallel SystemC and Processor Sleep Models. In *Proceedings of the 9th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '17)*. ACM, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3023973.3023975

[29] J. H. Weinstock, C. Schumacher, R. Leupers, G. Ascheid, and L. Tosoratto. 2014. Time-decoupled parallel SystemC simulation. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–4. https://doi.org/10.7873/DATE.2014.204

[30] Éder F. Zulian. 2017. gem5.TnT: Tips and tricks to make your life easier when dealing with gem5. https://github.com/tukl-msd/gem5.TnT.

Static/Dynamic Real-Time Legacy Software Migration - A Comparative Analysis

Irune Yarza, Mikel Azkarate-Askatsua, Peio Onaindia, Philipp Ittershagen, Kim Grüttner and Wolfgang Nebel

Static/Dynamic Real-Time Legacy Software Migration – A Comparative Analysis

Irune Yarza
Mikel Azkarate-askatsua
Peio Onaindia
iyarza@ikerlan.es
mazkarate@ikerlan.es
ponaindia@ikerlan.es
Ikerlan Technology Research Centre,
Dependable Embedded Systems Area
Arrasate-Mondragón, Spain

Philipp Ittershagen
Kim Grüttner
philipp.ittershagen@offis.de
kim.gruettner@offis.de
OFFIS - Institute for Information Technology
Oldenburg, Germany

Wolfgang Nebel
nebel@informatik.uni-oldenburg.de
C.v.O. Universität Oldenburg
Oldenburg, Germany

ABSTRACT

Evolution to next generation embedded systems is shortening the obsolescence period of the underlying hardware. As this happens, software designed for those platforms (a.k.a., legacy code), that might be functionally correct and validated code, may be lost in the architecture and peripheral change. As embedded systems often have Real-Time (RT) computing constraints, the legacy code retargeting issue directly affects RT systems. Binary translation techniques have been widely applied for legacy code migration. However, there are just a few works that consider RT legacy code. Therefore, this paper presents a static and a dynamic binary migration approach (based on QEMU and Rev.ng respectively) and analyzes and compares their suitability as RT code migration solutions. The comparison shows that among the proposed solutions, the static is the most appropriate for short-running RT legacy code, since it ensures lower translation overhead and a more deterministic timing behavior. Instead, the dynamic approach might be a suitable solution for RT legacy code with long periods (over 0.01 s) and mainly composed of complex floating point computations, since the dynamic translation and optimization overhead is not that significant on long-running benchmarks and the static translation implies a great slowdown on benchmarks containing floating point operations.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

binary translation, legacy code, real-time systems, retargeting

ACM Reference Format:
Irune Yarza, Mikel Azkarate-askatsua, Peio Onaindia, Philipp Ittershagen, Kim Grüttner, and Wolfgang Nebel. 2020. Static/Dynamic Real-Time Legacy Software Migration – A Comparative Analysis. In *Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '20), January 21, 2020, Bologna, Italy.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3375246.3375257

RAPIDO '20, January 21, 2020, Bologna, Italy

ACM ISBN 978-1-4503-7777-5/20/01…$15.00
https://doi.org/10.1145/3375246.3375257

1 INTRODUCTION

Companies within the embedded systems industry are facing a relentless demand for increasingly stringent requirements such as better performance, increased dependability, and energy efficiency, while offering a cost-effective product within a reduced time-to-market. This transition to next generation embedded systems is being encouraged by the rapid development of computing architectures. As a consequence, the obsolescence period of embedded systems is being shortened and there is a need to deal with legacy code. Legacy code is characterized by some particular properties: it usually runs on obsolete Hardware (HW) which is slow and expensive to maintain [13], uses customized and deprecated toolchain(s), has no or outdated documentation [11], and it is essential for the company [2] since it comprises business knowledge [12].

Due to the fact that classical process models focus on the development stage of software life-cycle instead of operation-maintenance stages, the process of updating legacy systems is usually complex, error-prone, time-consuming and requires high cost investment. In response to this problem, research efforts have provided several solutions. Nonetheless, when it comes to legacy software migration, Binary Translation (BT) appears to be a standard approach, as the binary that runs on the legacy HW can be ported to a new HW platform without a considerable expense of time, effort and money.

Although BT has been successfully applied for legacy software migration, it is necessary to consider that when dealing with RT legacy code migration, not just the functional properties, but also the timing behavior has to be preserved. To the authors knowledge, Cogswell [4] and Heinz [7] are the only ones who considered timing on their proposed retargeting solutions. However, they have limitations regarding their portability. Therefore, industry still needs a low-overhead embedded RT software retargeting solution that can be easily ported to different source and target architectures.

In the direction to solve this problem, the overall goal of this research line is to enhance the latest low-overhead machine-adaptable static/dynamic BT tools with the ability to preserve the timing behavior on the translated binary. This will enable the migration of RT embedded legacy code to a new HW platform with guaranteed

RT performance. To this end, two approaches were considered: (1) a dynamic approach based on Quick EMUlator (QEMU) [1] and (2) a static approach based on Rev.ng [5].

As a first step on the research, in a previous publication [15] we analyzed the suitability of QEMU, for its use in a RT property conserving retargeting process. In the same vein, this article analyzes and compares the suitability of the static and dynamic approaches (on a chosen test/evaluation set-up) for their use in a RT property conserving retargeting process. Therefore, the main contribution of this paper is the construction of a test environment to check whether is better to choose a static or a dynamic approach to port a particular RT legacy binary to a new architecture being able to reproduce the timing behavior on the legacy Instruction Set Architecture (ISA). The detailed technical contributions of this paper are:

- A survey on existing static and dynamic code translation techniques heeding portability, embedded systems or RT legacy code.
- A description of the static and dynamic RT legacy code migration approaches.
- A feasibility study and comparative analysis of the described static and dynamic RT legacy code migration solutions.

The remainder of this paper is organized as follows. An overview of related work in the area of machine-adaptable static and dynamic code translation techniques for embedded systems is provided in Section 2. Then, Section 3 presents the proposed solution and describes the static and dynamic approaches considered on this research for RT legacy code migration. In order to perform a feasibility study of the proposed approaches, Section 4 describes the construction of a test framework with means for high-resolution execution time measurement of periodically triggered software. The obtained experimental results are thus evaluated on Section 5. Finally, Section 6 gives an outlook on future work and a conclusion.

2 RELATED WORK

BT techniques have been widely studied and developed in the last decades. Table 1 provides a summary of the related work reviewed in [15]. Whereas this section analyzes the related work with a focus on aspects such as portability, and RT legacy code and system-level code support. Reviewed work and this section

Table 1: Related work summary. Cross-platform BT tools are analyzed according the following four aspects: static/-dynamic translation, portability, and RT legacy code and system-level code support.

Name	Static/Dynamic	Machine-adaptable	RT legacy Code	User-/System-level
TIBBIT [4]	Static	-	✓	System-level
Heinz [7]	Static	-	✓	System-level
UQBT [3]	Static	Source & Target	-	User-level
UQDBT [10]	Dynamic	Source & Target	-	User-level
QEMU [1]	Dynamic	Source & Target	-	User- & System-level
DisIRer [8]	Static	Target	-	User-level
CrossBit [14]	Dynamic	Source & Target	-	User-level
Rev.ng [5]	Static	Source & Target	-	User-level
LLBT [9]	Static	Target	-	User-level

Development cost is one of the main concerns when developing a binary translator, since the implementation of such a system from scratch requires great effort. So given that BT tools are highly dependent on the source and target architectures, researchers adopted the general approach of portable compilers, where machine-dependent and machine-independent concerns are separated, to provide a **machine-adaptable** binary translator. The first machine-adaptable solution, UQBT [3], supports multiple source and target machines by using specifications that describe the ISA and Operating System (OS). Unlike UQBT, most machine-adaptable solutions provide multiple front-ends and benefit from a retargetable compiler to provide support for multiple target ISA.

When dealing with time sensitive code migration, not just its functional behavior, but also the timing behavior of time critical tasks has to be preserved. From the related work, just [4] and [7] presented a migration path for **RT legacy software**. The former, proposes a instruction level annotation approach that describes the amount of time required to execute the block on the source processor. This way a virtual clock is provided to the run-time system that compares its value to the target clock and enforces an equivalent timing behavior. This approach is efficient for simple architectures where the execution time of each instruction is predictable. The latter, implements static temporal barriers to reduce the runtime overhead of the delay computation. Based on a Worst Case Execution Time (WCET) analysis tool a set of delay constants are precomputed for each program point and according to the program context the appropriate value is selected at runtime.

Embedded systems often contain a significant amount of low-level code dedicated to control either processor integrated or external devices (e.g. Analog-to-Dygital Converter (ADC); serial, Ethernet or CAN controller; sensor/actuator), also known as **system-level code**. However, most of the approaches on the State of The Art (SOTA) propose migration solutions for **user-level code**, where the underlying OS's Application Programming Interface (API) abstracts the low-level code from the application. The approaches presented in [4] and [7] support system-level binaries, whereas, QEMU that was first designed for Linux machine emulation, now supports also system-level code.

3 RT LEGACY CODE MIGRATION

In order to provide a migration path to RT legacy software, two approaches have been considered: a dynamic approach based on QEMU [1] machine emulator and a static approach based on Rev.ng [5] binary instrumentation framework. The following subsections describe the characteristics of legacy system to be ported and present the assumptions and current constraints of the proposed migration solution. The proposed static and dynamic migration approaches are then described in detail, setting the focus on how each of the tools performs the BT process and how each of them overcomes the main portability aspects.

3.1 Legacy System

The left side of Figure 1 depicts the legacy system architecture, which follows the typical pattern of a reactive control system. The application is periodically triggered (at t_0, t_1, ...) to read new data from the sensors and update the actuators after a period of time

(dt). For an appropriate behavior of the system under control, the duration of the application (dt) must be below the execution period ($t_{n\ 1} - t_n$). Moreover, every action on the legacy application that implies information exchange (e.g. read/write from/to Input/Output (I/O) device buffer or shared variables) is also likely to be timing critical.

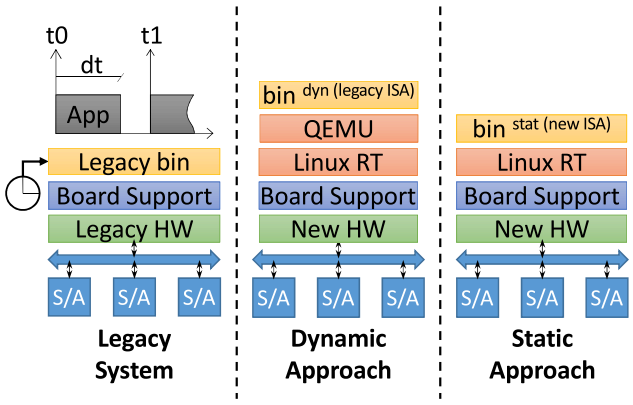

Figure 1: Runtime architecture of RT legacy code running on the legacy system (left hand), dynamic approach (middle), and static approach (right hand).

3.2 Assumptions and Constraints

The legacy code block that needs to be ported, is treated as a gray box that is being reused with little knowledge of its implementation. The source code is available, however, there is a preference to keep it unmodified due to possible unfamiliarity with the code and to increase the usability of the approach.

The current proposal only considers applications that have manually adapted I/O accesses to the new HW platform and do not make use of any HW timer (i.e. we currently only consider pure computational applications and do not consider I/O virtualization between the legacy and new HW platform). Moreover, the current approaches provide means to implement the periodic execution loop, but do not yet support time enforcement at a finer granularity. These constraints will be lifted in future work.

3.3 Dynamic Approach

The dynamic approach takes advantage of QEMU [1] to translate legacy code on run-time. As I/O and timer virtualization is not supported, in order to access the host timer and implement a periodic execution loop (without re-launching QEMU), the legacy application and QEMU's source code have to be adapted before translation. The adapted legacy application is then compiled for the legacy processor and runs on top of adapted QEMU. The translator, is launched on top of a minimal Linux distribution, which has been configured using the PREEMPT_RT patch [1]. The adapted QEMU is launched with the highest allowed priority [2]. In the center of Figure 1 the described dynamic approach run-time architecture is shown.

The following subsections describe how the legacy application is adapted, which is QEMU's translation process and how QEMU's source code is adapted to reach our goal.

3.3.1 Adapt legacy application. In the legacy application, I/O accesses are replaced with I/O variables and a particular approach is followed to periodically launch the application. An application container is defined, which initializes state variables (as it is done in the legacy application) and sets an infinite execution loop. Inside this loop, first an empty function call is inserted[3], `start_period()`, that will allow the identification of the period start point from QEMUś translation process. Then, input variables are updated (from csv file), the legacy application's behavioral part is executed in a run-to completion manner and the content of output variables is written back (to csv file). Finally, an empty function call is inserted, `end_period()`, to identify the end of the period from QEMU's translation process. Using empty function calls to annotate the legacy application we ensure that there is a branch in the code, consequently this instructions will be the first ones in their corresponding Translation Block (TB).

3.3.2 QEMU. The core element in QEMU is its code generator, Tiny Code Generator (TCG), which is responsible for the dynamic translation of target source code into host machine code. As a machine-adaptable Dynamic Binary Translation (DBT), TCG adopts the general approach in portable compilers. Therefore, the source code TBs are first translated into tiny code instructions, a machine independent Intermediate Representation (IR), and then this IR code is further translated into target machine code. Once translated, the TBs are stored in the code cache to be reused in future runs. TB caching reduces translation overhead since the time spent on code translation is reduced. For the sake of simplicity, when the code cache overflows, all stored TBs are removed. Moreover, to avoid returning control from the code cache to the emulation manager and back again to the code cache, QEMU chains consequentially executed TBs. As an example, after the execution of TB1, as there was no chaining, execution returns to the emulation manager. In that case, the next TB, TB2, has to be found, generated (if target machine code for this TB is not available), executed and chained to TB1. This way, the next time TB1 is executed TB2 will follow the execution without returning control to the emulation manager.

Figure 2 illustrates in a flow diagram QEMU's run-time behavior. Execution starts, and the first step is to set-up the Virtual Machine (VM) environment according to its specifications (e.g., number of CPUs, RAM size and available devices). Then, CPU execution starts with `cpu_exec()` function, referred to as the 'main execution loop'. Inside this execution loop, the first step is to handle the interrupts if any. Afterward, `tb_find()` function searches the next TB according to the current Program Counter (PC) value. If no TB is found, target machine code is generated through `tb_gen_code()` function, which subsequently call functions `gen_intermediate_code()` to translate source code into tiny

[1] The main purpose of PREEMPT_RT patch is to improve the RT behavior on Linux by reducing the kernel's scheduler latency and response time. Moreover, PREEMPT_RT achieves a more deterministic Linux environment without the need for a specific API.

[2] The highest allowed priority is 98, since PREEMPT_RT uses 99 as the priority for the kernel task sets and interrupt handler.

[3] An empty function call is a call to a function whose body is empty.

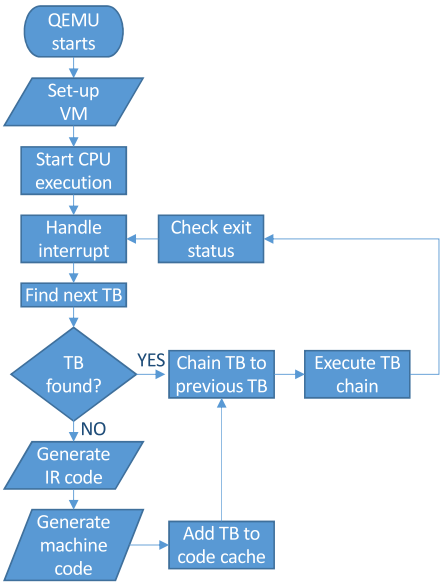

Figure 2: QEMU's DBT flow diagram.

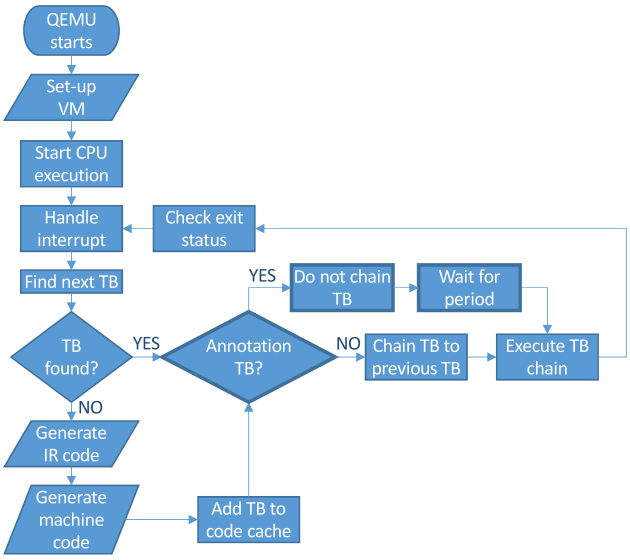

Figure 3: QEMU's DBT flow diagram adapted to establish the periodic execution loop.

code instructions and `tcg_gen_code()` to convert intermediate code into target machine code. After target machine code has been generated, the TB is stored in the code cache, `tb_jmp_cache`, at an index found by `tb_jmp_cache_hash_func()`. The generated/-found TB is then chained to the previous TB, `tb_add_jump()`, to avoid a context switch in a following run. Finally, translated code execution continues through `cpu_loop_exec_tb()` function.

3.3.3 Adapt QEMU. In order to establish the periodic execution loop, apart from the legacy application, QEMU's source code also has to be adapted in such a way that it identifies the annotations (empty function calls) in the legacy application and implements the periodic loop. Figure 3 illustrates in a flow diagram the run-time behavior of the adapted QEMU. QEMU identifies TBs with the PC value of the first instruction in the block. Since we annotated the legacy application with empty function calls, we ensured that these instructions will be the first instruction in the TB. So, if the generated/found TB corresponds to the annotation PC value (start_period_pc() or end_period_pc()) an auxiliary code is inserted that, based on Linux high resolution timers, gets the start/end time (saved in pStart_dyn or pEnd_dyn), measures the duration, compares it with the period and waits until they are equal. However, as QEMU chains consequent TBs, start and end TBs would be chained to previous TBs and it would not be possible to detect them. So, it is necessary to ensure that these TBs are never chained to the previous one.

3.4 Static Approach

The static legacy code migration approach employs Rev.ng [5], to translate a statically linked Linux binary into equivalent target machine code. As I/O and timer virtualization is not supported, the legacy application has to be adapted. First of all, accesses to I/Os are replaced with I/O variables. Then, an application container is defined, which initializes the state variables (as it is done in the legacy application) and creates a periodic execution loop that contains the adapted legacy application's behavioral part which is executed in run-to completion mode. The application container which includes the legacy code is compiled using a Linux toolchain for the legacy architecture. The statically generated binary is then translated off-line, before run-time. Once translated, the new binary runs with priority 98 (highest allowed) on top of a minimal Linux distribution that has been configured with the PREEMPT_RT patch. On the right hand of Figure 1 the run-time architecture of the static migration approach, as it has been described, is shown.

3.4.1 Rev.ng. Rev.ng is a binary analysis framework whose core element is, Revamb Static Binary Translation (SBT) tool, which combines the benefits of QEMU, with those of Low Level Virtual Machine (LLVM). LLVM is a compilation framework that provides source and target independent optimization support as well as resources for multiple machine code generation. The main components of LLVM's architecture are: (1) the front-ends, which translate source code in a variety of languages into LLVM IR. Clang, a C, C++ and Object-C front-end, is the one that has received the most attention; (2) Its IR, the core element in LLVM, a target-independent low-level programming language; (3) the Pass Framework, that is in charge of IR to IR transformation, most of the times seeking for code optimization and/or analysis; and (4) the back-end, which supports machine code generation for multiple instruction sets.

Rev.ng currently supports static ARM, MIPS and x86-64 Linux binaries as input and can generate machine code for X86-64 output architecture. However, even if the current tool suite supports just a few input/output architecture combinations, the fact that it is based

on QEMU and LLVM makes Rev.ng adaptable to other source/target architectures supported by QEMU [4] and LLVM [5] respectively.

As already mentioned, the core element in Rev.ng is its SBT tool. Revamb parses the statically linked Linux binary and uses QEMU's TCG as a front-end to generate tiny code instructions from any of the input architectures it supports. Then code in QEMU IR form is further translated into LLVM IR instructions. However, in QEMU, certain features such as syscalls and complex instructions (e.g. floating point division) are handled through a set of external functions (written in C) known as helper functions. Therefore, using Clang, QEMU helper functions are obtained in the form of LLVM IR and statically linked before generating the LLVM module. Besides the helper functions, additional support is needed mainly for initialization purposes. To this end, Revamb provides a set of support functions which are linked to the LLVM module. Then, the linked LLVM IR module is translated into machine code using LLVM compiler infrastructure. Figure 4 depicts the translation process of Rev.ng tool suite, which combines the use of QEMU's front-end and LLVM.

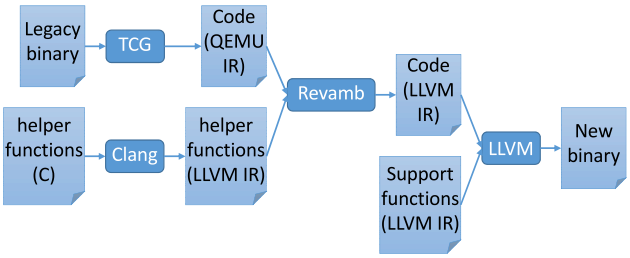

Figure 4: Rev.ng's SBT process combining the use of QEMU, Revamb and LLVM.

4 FEASIBILITY STUDY

The feasibility study assess the static and dynamic migration approaches described above with respect to timing. To do so, a test framework has been constructed, which provides means to measure the execution time of a selection of WCET representative benchmark programs, provided by the Mälardalen WCET research group [6] running on the legacy platform and on the new HW platform using both migration approaches, static and dynamic. The obtained results are then analyzed and compared in Section 5.

The test framework has been implemented on top of the following two Evaluation Boards (EBs): the ZC702 with a Zynq-7000 XC7Z020 SoC (consisting of a FPGA and an ARM Cortex-A9 processor with an operation frequency of 666 MHz) and the MinnowBoard Turbot Dual-Core with an Intel Atom E3826 processor with an operation frequency of 1463 MHz. The former is employed as the source processor (legacy), whereas the latter is used as the target processor where the static and dynamic legacy code migration techniques are tested [6]. To measure execution time on the ARM processor we used Xilinx's Board Support Package (BSP) to access the global timer counter, whereas to perform the measurements on the Intel Atom processor the Linux high-resolution timer has been used.

4.1 Dynamic Instrumentation

For the timing assessment of the dynamic approach, benchmarks need to be instrumented. However, the dynamic approach instrumentation solution is a twofold technique. On the one hand, the source code is annotated with empty function calls (`start_time()` and `end_time()`) to ease the start/end detection in QEMU. Using an empty function call we ensure that there is a branch in the code, consequently this instruction will be the first in the TB and we will be able to detect it though the PC. On the other hand, QEMU source code has been modified to integrate start/end PC detection (`start_time_pc` and `end_time_pc`) and perform the execution time measurements. When launching QEMU, the `start_time_pc` and `end_time_pc` values corresponding to the running benchmark are passed through arguments. The function in charge of finding the next TB, `tb_find`, identifies `start_time_pc` and `end_time_pc` and computes the duration. Moreover, as previously mentioned, QEMU chains consequently executed TBs to avoid context switch cost. As a consequence, start and end TBs would be chained to former TBs and control would not return to the execution manager. Therefore, `tb_find` function has been altered to avoid start and end TB chaining.

5 FEASIBILITY RESULT ANALYSIS

The feasibility survey compares the execution time of the Mälardalen WCET benchmarks [6] running on top of the legacy HW platform and the new HW using both, dynamic and static migration approaches. Given that the selected benchmarks contain a great variety of algorithms (including loops, nested loops, use of array and/or matrices and use of floating point operations), we get a wide analysis of the timing behavior of the proposed static and dynamic solutions.

5.1 Platform configuration

For the execution time analysis on the legacy HW (ZC702), the Vivado Zynq example project is used. The generated bitstream is exported to Software Development Kit (SDK), where benchmarks are compiled (without any optimization [7], -O0).

The same example project is used to run the legacy code on the new HW platform (MinnowBoard) through the dynamic approach. However, as explained before, due to QEMU's start-up procedure, the code has to be compiled and linked to be placed at the OS starting memory location, which in the case of armv7 architecture is 0X10000. As well as for the legacy HW, the benchmarks have been compiled without optimization.

[4] QEMU supports the emulation of various architectures including: Alpha, ARM, CRIS, x86, MicroBlaze, MIPS, OpenRISC, PowerPC, RISC-V, SH4, Sparc and their 64-bit variant when applicable.
[5] LLVM's back end supports many ISAs, including ARM, MIPS, PowerPC, Sparc, x86 and x86-64. However, just x86 (both 32-bit and 64-bit), ARM and PowerPC include most of the features.

[6] Despite the fact that the Cortex-A9 processor is not a legacy HW platform, it has been chosen for the feasibility analysis for the fact that it is supported by the selected SBT tool. However, Rev.ng can be inexpensively adapted to support other source/target ISAs.
[7] This is a common practice in RT systems where the WCET is important for reliability or correct functional behavior

Regarding the static legacy code migration approach, Rev.ng provides a cross-toolchain for each of the supported input architectures. Therefore, the statically linked Linux input binary has been generated using the corresponding toolchain (armv7a-hardfloat-linux-ublibceabi-gcc) and without applying any optimization. Then, the input binary has been statically translated using the *translate* script provided with Rev.ng tool suite.

5.2 Evaluation process and results

To get the results, each benchmark has been executed 15000 times (statistically relevant enough) on the legacy and new HW platforms (using dynamic and static migration solutions) while collecting timing data. Given that we are targeting the migration of a reactive control system where the application is periodically triggered, QEMU is launched only once executing the benchmarks periodically and the first benchmark runs are excluded from the analysis. This way the DBT warm-up time, code translation/optimization overhead on the first runs when there is still no translated code available in the code cache, does not affect the measurements[8]. Together with the WCET benchmarks an empty application[9] has also been analyzed. Measuring the empty application execution time provides means to measure the overhead introduced by the underlying system on either migration approach, which is composed of: QEMU and Linux PREEMPT_RT on the dynamic solution and the extra instructions inserted on the code by Rev.ng translator and Linux PREEMPT_RT on the static solution.

5.2.1 Translation overhead analysis. The translation overhead analysis is performed based on the empty application, measuring the execution time on the legacy and new ISAs following a dynamic and static translation process. Figure 5 shows the collected data distribution with a zoom in the maximum execution time result area. Table 2 contains the minimum, maximum, average, standard deviation and 99%-quantile of the collected data.

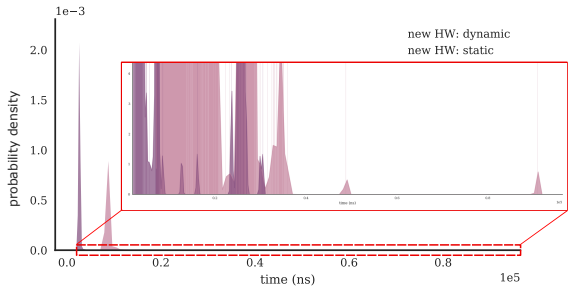

Figure 5: Distribution of execution time data collected running the empty application on the new HW platform following a dynamic/static translation process.

Results show that, as expected, the average translation overhead of the DBT solution is higher than the average static approach overhead, almost 3.7x greater. In the dynamic approach, translation and optimization counts on the measured execution time and even though QEMU applies counter measures, such as translated code caching and consequent TB chaining, it still implies great overhead. Whereas the static approach is capable of generating more efficient code, since neither translation nor optimization counts on the execution time. Therefore, it is possible to apply more aggressive optimizations.

Table 2: Maximum, minimum, average, standard deviation and 99%-quantile of the measured execution time when running multiple times the empty application on the new HW platform following a dynamic/static translation process.

	Execution time (ns)				
	min	**max**	**avg**	**std**	**99%-quantile**
Dynamic	8342	358739	10043,33	5913,59	38860,76
Static	2558	330480	2751,55	1415,85	2985,18

Regarding the 99%-quantile, which indicates the value bellow which the 99% of the measured values are found, the difference between the static and dynamic migration approaches is even greater. The 99%-quantile in the dynamic migration approach is 13x higher than that in the static approach and 3.9x higher than the dynamic average execution time. Whereas the 99%-quantile in the static approach is just 1.1x higher than the static average execution time. The standard deviation is similar in both migration approaches, 58.9% of the average in the dynamic vs. 51.5% in the static approach. These results lead to the conclusion that although both approaches have little difference on the maximum execution time, these sporadic corner execution time values, which can be appreciated in the zoom-in area in Figure 5, are more frequent in the dynamic migration approach. This is reflected on the 99%-quantile, which greatly differs form the average.

To get a better knowledge about how each migration approach performs depending on the characteristics of the translated binary, the following subsection provide a Static vs. Dynamic re-targeting comparative analysis.

5.2.2 Static vs. Dynamic migration. The comparative analysis is performed based on the execution time results obtained from running a WCET representative benchmark suite on top of the legacy and new HW platforms. The benchmarks are first compiled for the legacy architecture and then translated following static and dynamic migration approaches.

In order to solve scaling problems, results have been clustered into 4 different graphs, see Figure 6. These graphs show a comparison between the average value and 99%-quantile (overlapped) of the measured execution time on the new ISAs. Moreover, the standard deviation is represented as an error bar on the average value. Each graph shows the timing results obtained for the dynamic and static migration approaches.

When analyzing the results, benchmarks are classified into short- and long-running according to their average execution time on the legacy HW. We consider a benchmark to be short-running bellow 100000 ns and long-running over 100000 ns (measured on the legacy processor). Moreover, for a better result analysis, benchmarks are classified according to their characteristics (see Table 3). Based

[8] An analysis of the (quantified) performance during this warm-up is out of the scope of this work.
[9] We consider an empty application that whose main function does not contain any instruction.

on the information provided by the Mälardalen WCET research group [6], we have classify benchmarks depending on the type of operations they contain: (1) complex computations, (2) simple computations or (3) control flow statements. The first group is expected to have a low translation overhead, since the new processor can handle better complex computations. The second group also, since the translator can efficiently translate this code. Whereas the third group is expected to have a high translation overhead, since control flow statements hinder translation efficiency, mainly in the dynamic approach due to the difficulties to apply TB-chaining, but also in the static approach because statements might depend on run-time behavior.

Table 3: Benchmark classification. S = always single path program. L = contains loops. N = contains nested loops. A = uses arrays and/or matrixes. B = uses bit operations. R = contains recursion. U = contains unstructured code. F = uses floating point calculation. CC = composed of complex computations. SC = composed of simple computations CF = composed of control flow statements.

Benchmark	S	L	N	A	B	R	U	F	CC/SC/CF
adpcm	-	✓	-	-	-	-	-	-	CF
bs	-	✓	-	✓	-	-	-	-	CF
cnt	-	✓	✓	✓	-	-	-	-	CF
compress	-	✓	✓	✓	-	-	-	-	CF
cover	✓	✓	-	-	-	-	-	-	CF
crc	✓	✓	-	✓	✓	-	-	-	CC
duff	✓	✓	-	-	-	-	✓	-	CF
edn	✓	✓	✓	✓	✓	-	-	-	CC
expint	✓	✓	✓	-	-	-	-	-	CF
fac	✓	✓	-	-	-	✓	-	-	CF
fdct	✓	✓	-	✓	✓	-	-	-	CC
fft1	✓	✓	✓	✓	-	-	-	✓	CC
fibcall	✓	✓	-	-	-	-	-	-	CF
fir	-	✓	✓	✓	-	-	-	-	SC
insertsort	-	✓	✓	✓	-	-	-	-	SC
janne_complex	✓	✓	✓	-	-	-	-	-	CF
jfdctint	✓	✓	-	✓	-	-	-	-	SC
lcdnum	-	✓	-	-	✓	-	-	-	CF
lms	✓	✓	-	✓	-	-	-	✓	CC
ludcmp	-	✓	✓	✓	-	-	-	✓	CC
matmult	✓	✓	✓	✓	-	-	-	-	SC
minver	✓	✓	✓	✓	-	-	-	✓	CF
ndes	-	✓	-	✓	✓	-	-	-	CC
ns	-	✓	✓	✓	-	-	-	-	CF
prime	✓	✓	-	-	-	-	-	-	SC
qsort-exam	-	✓	✓	✓	-	-	-	✓	CF
qurt	✓	✓	-	✓	-	-	-	✓	CC
recursion	✓	-	-	-	-	✓	-	-	CF
select	-	✓	✓	✓	-	-	-	✓	CF
sqrt	✓	✓	-	-	-	-	-	✓	CC
st	✓	-	✓	-	✓	-	-	✓	CC
statemate	-	✓	-	-	-	-	-	-	CF

Results show that most of the benchmarks that have been analyzed run faster applying the static translation approach (4.6x faster on average), which might be due to the following two reasons: (1) the dynamic approach has heavy run-time overhead, including code translation/optimization and run-time management; and (2) due to the fact that optimization time counts on execution time, the dynamic approach does not apply aggressive optimization, which leads to worse code quality. However, we did not find any relationship between the benchmark characteristics and the average dynamic/static execution time ratio. As a general rule, the shorter the benchmark execution time, the higher the 99%-quantile/average ratio in the dynamic approach, which goes from 1.03 on long-running to 3.74 on short-running benchmarks. In fact, QEMU's run-time overhead is significant on short-running benchmarks, whereas it is not so, or at least not that much significant, on long-running benchmarks. Moreover, from analyzing the average execution time ratio between dynamically translated binaries running on the new HW and legacy binaries running on the legacy HW, it can be appreciated that the 10 slowest benchmarks (bs, fac, fdct, fibcall, janne_complex, lcdnum, minver, qsort_exam, select, statemate) are mainly composed of control flow statements and simple computations, except for fdct. Whereas from the analysis of the average execution time ratio between statically translated binaries running on the new HW and legacy binaries running on the legacy processor, it can be appreciated that among the 10 slowest benchmarks (bs, expint, fft1, janne_complex, lcdnum, lms, ludcmp, qurt, sqrt, st), some are mainly composed of control flow statements and little computations (e.g., bs, lcdnum), but many others are mainly composed of complex computations (e.g., fft1, lms, sqrt, st). However, these slow benchmarks mainly composed of complex computations, share a common characteristic: they all contains floating point operations. In fact, benchmarks with complex floating point operations (e.g., fft1, lms, ludcmp, sqrt, st) have the lowest average dynamic/static ratio.

6 CONCLUSIONS AND FUTURE WORK

This work aimed at describing the proposed static and dynamic embedded RT legacy code migration approaches and at performing a feasibility analysis of both solutions. The dynamic approach is based on QEMU machine emulator, whereas the static approach is based on Rev.ng binary instrumentation framework. When migrating a RT application, not just its functional properties but also the timing behavior needs to be preserved. Therefore, the feasibility study compares both migration approaches with regards to measured execution time. To this end, a test framework has been constructed, which provides means to measure the execution time of code ported using both migration approaches. The test environment has been implemented on top of an Intel Atom E3826 processor, where a selection of WCET representative benchmarks compiled for ARM Cortex-A9 have been ported. From the experimental result analysis, it can be concluded that among the proposed migration approaches, the static is the most appropriate method to port short-running RT legacy code. Whereas the dynamic approach might be a better choice when porting RT legacy code with long periods (over 0,01s).

As already mentioned in the introduction, this work described early results. Future work will provide means to preserve the timing behavior of the legacy code on the new HW platform. To this end, it is necessary to define the timing constraints that the system has to meet and their granularity and the execution time control mechanism that will be integrated in the new ISA. The timing enforcement solution together with I/O virtualization implemented on an appropriate BT system (dynamic or static, depending on the characteristics of the legacy software to be ported) will provide means to enforce the legacy timing requirements on the new HW platform and a way of time sensitive interaction between the control system and the external environment.

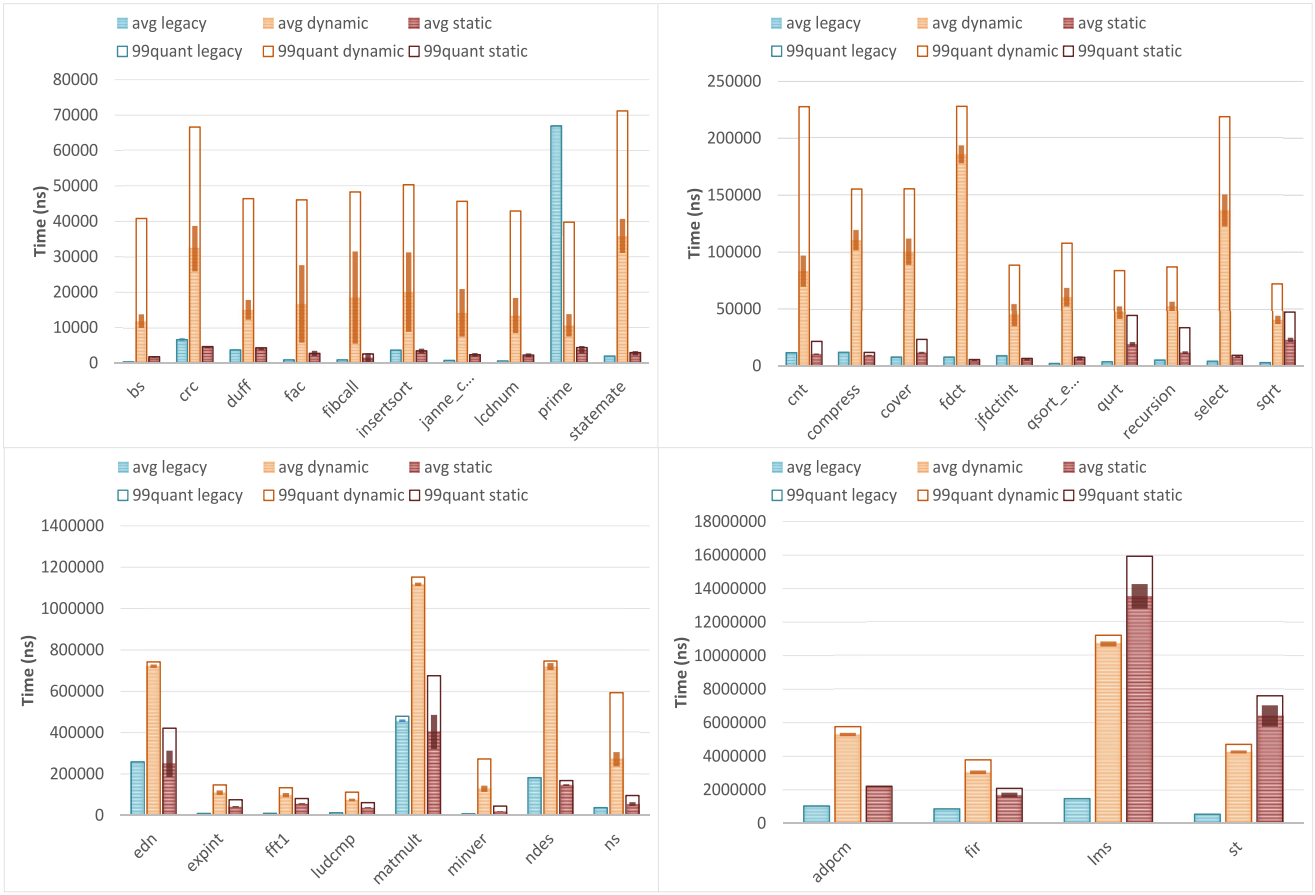

Figure 6: Timing results of benchmarks running on the legacy and new HW platforms: static vs. dynamic translation.

7 ACKNOWLEDGMENTS

The authors would like to thank the Rev.ng tool suit developers for supporting them with the Rev.ng tool and providing them access to their private repository.

REFERENCES

[1] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In USENIX Annual Technical Conference, FREENIX Track. 41–46.
[2] Keith Bennett. 1995. Legacy systems: Coping with success. IEEE software 12, 1 (1995), 19–23.
[3] C. Cifuentes and M. Van Emmerik. 2000. UQBT: adaptable binary translation at low cost. Computer 33, 3 (2000), 60–66. https://doi.org/10.1109/2.825697
[4] Bryce Cogswell and Zary Segall. 1995. Timing insensitive binary to binary translation of real time systems. In Workshop on Architectures for Real-Time Applications, ISCA.
[5] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In CC 2017. ACM, 3033028, 131–141. https://doi.org/10.1145/3033019.3033028
[6] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET Benchmarks – Past, Present and Future. In WCET2010, Björn Lisper (Ed.). OCG, Brussels, Belgium, 137–147.
[7] Thomas Heinz. 2008. Preserving temporal behaviour of legacy real-time software across static binary translation. In Proceedings of the 1st workshop on Isolation and integration in embedded systems. ACM, 1–4.
[8] Yuan-Shin Hwang, Tzong-Yen Lin, and Rong-Guey Chang. 2010. DisIRer: Converting a retargetable compiler into a multiplatform binary translator. ACM Transactions on Architecture and Code Optimization (TACO) 7, 4 (2010), 18.
[9] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. 2012. LLBT: an LLVM-based static binary translator. In Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems. ACM, 51–60.
[10] David Ung and Cristina Cifuentes. 2000. Machine-adaptable dynamic binary translation. In ACM SIGPLAN Notices, Vol. 35. ACM, 41–51.
[11] Christian Wagner and Christian Wagner. 2014. Model-Driven Software Migration. Springer.
[12] M. Wahler, R. Eidenbenz, C. Franke, and Y. A. Pignolet. 2015. Migrating legacy control software to multi-core hardware. In Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on. 458–466. https://doi.org/10.1109/ICSM.2015.7332497
[13] Bing Wu, Deirdre Lawless, Jesus Bisbal, Jane Grimson, Vincent Wade, Donie O'Sullivan, and Ray Richardson. 1997. Legacy system migration: A legacy data migration engine. In Proceedings of the 17th International Database Conference (DATASEM'97). 129–138.
[14] Yindong Yang, Haibing Guan, Erzhou Zhu, Hongbo Yang, and Bo Liu. 2010. Crossbit: a multi-sources and multi-targets DBT.
[15] Irune Yarza, Mikel Azkarate-askasua, Kim Grüttner, and Wolfgang Nebel. 2018. Real-Time Capable Retargeting of Xilinx MicroBlaze Binaries using QEMU: A Feasibility Study. In Proceedings of the Rapido'18 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools. ACM, 3180671, 1–8. https://doi.org/10.1145/3180665.3180671

Towards A Power Advisor in a Devkit for Internet-of-Things Microcontrollers

Vincent Morice, Florence Maraninchi and Jérôme Cornet

Towards A Power Advisor in a Devkit for Internet-of-Things Microcontrollers

Vincent Morice*
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France
vincent.morice@univ-grenoble-alpes.fr
vincent.morice@st.com

Florence Maraninchi
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France
florence.maraninchi@univ-grenoble-alpes.fr

Jérôme Cornet
STMicroelectronics, 38019, Grenoble, France
jerome.cornet@st.com

ABSTRACT

Microcontrollers (MCUs) for the Internet-of-Things (IoT) are powerful and versatile computing platforms, which may be hard to program correctly and efficiently; power performance is particularly important. We inestigate automatic methods to detect software performance anti-patterns for this class of systems, so as to help the software developer with power-related aspects. We use a virtual prototype, i.e., we execute the real object code on a simulated model of the hardware platform, given as a transaction-level model (TLM) augmented with dedicated monitors. We study two cases taken from an industrial example, and show that our method can help detect patterns that would be difficult to detect statically, even when the source code is available, because they involve the state of the hardware and the timing of operations.

KEYWORDS

Transaction-Level Modeling, IoT, Power Consumption, DevKit

1 INTRODUCTION

1.1 IoT Chips

The STM32WB (STMicroelectronics), the CC2540 (Texas Instrument), the BM70 (Microchip), or the QN9080 (NXP), are powerful and versatile microcontrollers units (MCUs), containing one or more CPUs (like ARM-Cortex series) and integrated with various sensors, actuators, and connectors, designed with power performance in mind. They may be powered by a battery and offer sophisticated clock and power modes. The difficulty to write good software for those platforms is a known problem in the industry: software should exploit the capabilities of the hardware correctly (respecting its specification), efficiently (consuming as little power as possible), and securely (resisting attacks). In this paper we focus on power efficiency. Dealing with numerous sensors makes this type of HW/SW platform a cyber-physical system (CPS): some knowledge about the physical environment can help build efficient solutions, for instance by selecting appropriate refreshment rates for sensors. A thorough understanding of the computing platform is needed in order to guarantee power performance, because a very small variation in the software can have tremendous effects on the consumption, and therefore the lifetime of the IoT object. In one of the examples below (see 2.1.2), the lifetime increases from approximately 3 days to approximately 8 months with a 700 mAh battery.

*Also with STMicroelectronics, 38019, Grenoble, France.

1.2 Methods and Tools for the Software Developer

Difficulties to write efficient and correct software can be tackled in various ways. Let us consider the pros and cons of each solution.

(1) Careful reading of the source code by experts: requires high expertise, not necessarily available at the customer site, and access to the source code, not always available for libraries.

(2) Using some Hardware-Abstraction-Layer (HAL) provided by the vendor: helps avoiding bugs and writing portable code; but it is sometimes counter-productive as far as power efficiency is concerned.

(3) Using interactive debugging tools like GDB for the software running on the real chip: requires heavy human interaction and may fail to correct power consumption or timing problems because they are due to the invisible hardware states.

(4) Executing the software on some simulated (thus fully observable) model of the hardware: gives non-intrusive access to many details of the hardware platform that may be hard to observe on the real chip, while having a clear impact on power consumption, like the traffic on the bus or the power modes of the CPU; some vendors provide simulators for a partial view of their platform, like the configuration of the clocks.

(5) Applying data mining techniques to the automatic detection of patterns in execution traces [10]: relies on the availability of a large number of traces, and algorithms to search those large data sets; it is not meant for a monitoring (incremental) implementation.

(6) Automatically detecting software (anti-)patterns [14]: can identify critical points quickly in a large number of code lines, but these points are not necessarily bugs or actual problems in real executions; it was first proposed for object-oriented development; it is more difficult on less-structured low-level code.

1.3 Exploiting Virtual Prototypes

We focus on power performance: the MCU or the sensors provide several modes that influence power, which makes it very hard to talk about power at a syntactical level; we need to look at the dynamic behavior.

Our goal is to explore a solution of type (4) above. It relies on a virtual prototype of the platform, given as a Transaction-Level Model (TLM). Virtual prototypes in the form of TL models started almost 20 years ago as tools to help hardware and software developers communicate, and also to allow for developing software before the hardware is indeed available [8]; they are on the path to be a

key aspect of the IoT ecosystem, providing developers with tools that help accelerate software development [11].

Our ultimate objective is to provide a development kit based on the TL model of a platform, extended with performance diagnosis capabilities. This kit will automatically warn the user about inefficient uses of the hardware, at execution time. Our aim is not only to diagnose inefficient uses, but also to provide the user with some advice for better code, hence smoothing the learning curve.

1.4 Contributions and structure of the paper

In Sections 2 and 3 we study a weather station based on a STM32F411 chip [5]. The code is a refactoring of a representative real industrial example from STMicroelectronics and partially uses the hardware abstraction layer provided by the company. The chip is mounted on a Nucleo-64 prototyping board [3] extended with a sensor expansion board including various sensors [1]. We identify two classes of very common problems to be detected by monitors: polling loops, and inefficient use of sensors. we show why dynamic information is key to the detection of *real* problems, and we show how to choose specific events and patterns to observe in TL traces in order to find occurrences of those two problems. In Section 4 we give a quick view of the monitor implementation, and evaluate our solution. Sections 5 and 6 give related work and conclude.

```
void HAL_Delay(__IO uint32_t Delay) {
  uint32_t tickstart = 0;
  tickstart = HAL_GetTick();
  while((HAL_GetTick() - tickstart) < Delay) {}
}
uint32_t HAL_GetTick(void) { return uwTick; }
void HAL_IncTick(void) { uwTick++; }
void SysTick_Handler(void) { HAL_IncTick(); }
```

Figure 1: A delay implemented by a polling loop

```
int main(void) {
  init();
  while(1) {
    int16_t temp = EnvSensor_GetTemperature();
    int32_t pressure = EnvSensor_GetPressure();
    uint16_t humidity = EnvSensor_GetHumidity();
    printf("T, P, H = %d\n",temp,pressure,humidity);
    HAL_Delay(800);
}}
```

Figure 2: `main()` function of the weather station application

```
/* asking access to registers */
CLEAR_BIT(RTC->CR, RTC_CR_WUTE);
/* Wait till access to wakeup timer is allowed */
while(READ_BIT(RTC->ISR,RTC_ISR_WUTWF) != 1);
config_wakeup_timer();
```

Figure 3: Polling loop on registers of the wakeup timer

2 CASE STUDY: POLLING LOOPS

The first example is about detecting polling loops because it is generally more costly than waiting for an appropriate interrupt if available. When using methods based on the structure of the code, it means detecting the pattern `while(condition){};`, either syntactically, or on the binary code. As far as power consumption is concerned, the cost comes from the CPU being active (while it could be sleeping), but also from the bus activity induced by the evaluation of the condition. We aim at detecting polling loops even if the corresponding code is spread across several layers of low-level code, part of it possibly available in binary form only. Moreover, we would like to warn only when they have a significant effect, which may depend on the effective number of passes in the loop: looping twice is probably harmless, but 50 times may be significant.

We give two examples: the first one has to be replaced by something more efficient; the second one can be kept as it is.

2.1 Example: a Delay implemented by a Polling Loop

2.1.1 Existing Code. In the first example, polling is used to wait for a certain delay, as shown on Figure 1. `HAL_Delay()` takes as a parameter the time to wait (`Delay`), saves the current time and starts the polling loop. `uwTick` is a global variable incremented on every `SysTick` timer interrupt. The timer is configured to fire its overflow interrupt every millisecond. This example is used in a weather station application whose `main()` function is shown on Figure 2: it reads the sensors, then prints the value using the UART and then waits for 800 ms using the `HAL_Delay()` function.

2.1.2 Proposed Modification. Implementing the wait functionality efficiently on our platform is easy: put the MCU in a deep sleep mode and use the Real Time Clock (RTC) to wake it up. The RTC behaves as a timer, sending an interrupt after counting up to a certain value. The delay function now simply launches the count and puts the chip in STOP mode. The RTC also has to be initialized and its interrupt handler has to acknowledge the interrupt.

The STOP mode disables a lot of peripherals, including the MCU, the SysTick timer and the bus, in order to save power. According to the ST tool "STM32 CubeMX" [4], which gives approximate power consumption related to those MCU states, implementing the wait process using the RTC and the STOP mode instead of the polling loop decreases the consumption from 8.5 mA to 0.12mA. When powered by a 700 mAh battery, the code modification can extend the lifetime of the system approximately from 3 days to 8 months.

2.2 Example: Unlocking Registers

2.2.1 Existing Code. The second example (Figure 3) is a piece of code used to *unlock* access to the configuration registers of the wake-up timer. The manual [5] explicitly requires the software to implement this polling loop. The loop waits for the Wakeup Timer Writing Flag (`WUTWF`) to be set, indicating that other configuration registers can now be written. The Wakeup timer is a feature of the RTC component, so the left part of the condition checks for the `WUTWF` bit of the RTC Initialization and Status Register (`ISR`) using the macro `READ_BIT(RTC->ISR,RTC_ISR_WUTWF)`.

In the real platform the wakeup timer has its own clock, different from that of the CPU, and the unlocking operation takes a few cycles of the wakeup timer's clock. The effective number of iterations depends on the relative speeds of the CPU's clock and the wakeup timer's clock. The faster the CPU's clock, the more loop iterations can be executed during the time it takes to perform the unlocking operation.

2.2.2 Keeping the Polling Loop. This is a typical case of a harmless polling loop because the way the hardware is designed ensures a very small number of iterations (in our example, we observed 20 in the worst case). We use transaction-level models, which can be approximately timed with realistic assumptions on the duration of transactions. Hence simulations with such models also show a small number of iterations. This type of polling case can be ignored easily, by using a threshold on the number of iterations.

2.3 Characterizing and Detecting Polling Loops on TL Models

The first step is to characterize problematic polling loops by observing the dynamic behavior of the code. The idea is to focus on the *actual effect* of a polling mechanism, even if it is not written as a typical polling loop; or it is, but with harmless effects.

The effect of a polling mechanism is characterized by some repetitive pattern being observed, and we can decide to issue a warning after a given number of repetitions. We propose to observe the sequence S of READ transactions issued on the bus. Section 4 explains what is indeed observable (not all variable accesses do generate bus transactions), and how to filter out other READ transactions (e.g., from interrupt handlers).

In the first example above, there are 3 variables in the condition of the loop: `Delay`, `uwTick` and `tickstart`. The latter is stored in a CPU register, hence does not generate bus transactions. There are only two addresses that appear in READ transactions; (`HAL_GetTick()` function calls also generate READ transactions, but this is not an issue as explained in section 4.) `uwTick++` is filtered since it is executed in an interrupt handler.

In the second example there is only one access to a register of the RTC component that generates a READ transaction on the bus. Section 4 explains the instrumentation of the TL model, and shows how to detect the first case, while ignoring the second one.

The last important point is to characterize the repetitive patterns of READ transactions that are indeed issued on the bus for typical polling loops. For instance, if the code indeed contains a loop `while(condition){};`, each evaluation of the condition generates successive READ transactions, depending on the logical structure of the condition (in C the evaluation of `cond1 && cond2` does not evaluate `cond2` — hence does not access its variables — if `cond1` is false). There is no cache between the CPU and the bus, so each access to a variable generates a transaction. Observing repeated accesses to the variables of `cond1` is a hint that some polling situation might be involved, but it depends on which other accesses are observed between the accesses to the variables of `cond1`. Extracting also the variables of `cond2` helps confirm that there is a polling case.

2.4 Formalizing Patterns

As an example, let us consider the dynamic behavior of a program that checks repeatedly a condition `cond`:
`((a < 12)&&(a >= 0))||(b == 0)||(c == 0)`,
either written exactly like that, or obtained with macros and calls to other software layers. Notice the same variable may appear several times in the condition.

The iterative evaluation of `cond` generates sequences of accesses of the form $(\mathtt{a|aa|aab|aabc})^*$ if all variables are observable (i.e., generate transactions on the bus) or simply $(\mathtt{a|aa|aac})^*$ if, e.g., `b` does not.

Since we do not know in advance which addresses `a, b, c` to look for when searching for polling loops, our problem is a *parametric* version of the above example: we search for $(\mathtt{x|xy|xyz|xyzt|...})^*$, where `x, y, z, t, ...` can be instantiated with any address. The number of these "parameters" depends on the condition. It is not too restrictive to consider that it is *bounded* by a relatively small number P. In the sequel, we take $P = 3$ as an example (3 observable accesses). The problem becomes to check whether an execution trace is of the form: $(\mathtt{x|xy|xyz})^*$ for some `x, y, z`. For a given vocabulary V and $m \in V^*$, we can define $m \in (\mathtt{x|xy|xyz})^*$ as: $\exists (a,b,c) \in V^3 . m \in (\mathtt{a|ab|abc})^*$. If the vocabulary V where `a, b, c` belong is finite, this can be written as a simple regular expression, because $\exists$ can be expanded into ordinary alternatives: $m \in (\mathtt{x|xy|xyz})^* \iff m \in \bigvee_{(a,b,c)\in V^3} (\mathtt{a|ab|abc})^*$.

Finally, we replace the * by $^{[n,+\infty]}$, to start warning about the presence of a polling loop only if it exceeds a number n of effective iterations. In order to validate quickly the idea of using monitoring techniques to detect polling loops, we implemented a simpler (yet very frequent) case without Boolean operations. Instead of searching instances of $(\mathtt{x|xy|xyz})^{[n,+\infty]}$, we search for $\mathtt{x}^{[n,+\infty]}$ | $(\mathtt{xy})^{[n,+\infty]}$ | $(\mathtt{xyz})^{[n,+\infty]}$. See details in Section 4.

This example does not fully characterize all polling loops. For instance, instructions in the condition or the loop body (especially branching) can partially hide the repetition or produce a large number of repetitive addresses observed on the bus. Moreover, if we choose a very big P, we might end up detecting the infinite loop of the main program. However we think this is a promising approach, since it already detects a lot of typical polling loops in typical code samples. Further work will include more cases.

3 CASE STUDY: TEMPERATURE SENSOR

The second case-study concerns the use of the LPS22HB temperature sensor [2], which has several operating modes: (1) In **one-shot** mode the software has to set the one-shot bit to ask for a new value to be measured and prepared; the bit will be cleared by hardware when the new value is available; (2) In **auto-refresh** mode the sensor can produce a new value with a given period chosen in {13, 20, 40, 100, 1000} milliseconds. If the application needs less than one value per second, or more than one every 13 ms, it should use the one-shot mode. In both modes the last measure is always available so that the software can read the value at any time.

The idea of this example is to detect the uses of the one-shot mode that could be improved by using the auto-refresh mode instead. The one-shot mode can have a significant impact on the consumption, because it involves more transactions on the MCU bus: (1) the software sends the address of the control register of the sensor and the data ("1" on the one-shot control bit), (2) it sends the address of the temperature data register in order to read the value. The auto-refresh mode is far more efficient, since only step (2) is needed. Moreover a READ needs two distinct accesses to two 8-bit registers containing the MSBs and the LSBs of the temperature value.

3.1 Example

3.1.1 Existing Code. Our application uses the one-shot mode (see 4). The bit is set (`Set_One_Shot`), requiring a new measure; next time the function will be called, it will access the value (`Get_Temp`) and get this new measure. In our example, the function is called approximately each 800ms (see main code on Figure 2).

```
int16_t EnvSensor_GetTemperature(){
    int16_t SensorValue;
    if(TEMP_SENS_IsInitialized()){
      /*Read the previous value of the sensor
        and restart the One Shot for the
        next measurement*/
      if(TEMP_SENS_Get_Temp(&SensorValue)){
        if(TEMP_SENS_Set_One_Shot()){
           return SensorValue;}}}
    return -1;} // Error
```

Figure 4: Temperature measurement code in one-shot mode (simplified).

```
int16_t EnvSensor_GetTemperature(){
    int16_t SensorValue;
    if(TEMP_SENS_IsInitialized()){
      if(TEMP_SENS_Get_Temp(&SensorValue)){
         return SensorValue;}}}
    return -1;} // Error
```

Figure 5: Temperature measurement code in auto-refresh mode (simplified).

3.1.2 Proposed Modification. The alternative implementation is to use the auto-refresh mode, as shown on Figure 5. The auto-refresh period is set in the `init()` function called at startup in the `main()` function. The idea is to choose a period close to the delay D implemented by the application ($800ms$ in our case). If we do not have the full source code, or because it is hard to analyze statically anyway, D can be estimated dynamically by looking at the TL traces. On this example we choose $100ms$.

The cost of one bus transaction alone is very hard to estimate, so the power consumption of the `main()` function (Fig. 2) has been measured on the board with the initial and modified versions of the `EnvSensor_GetTemperature()` function and our proposed modification for the `HAL_Delay()` function (see section 2.1.2). Average consumption is 1.57 mA (± 0.01 mA) for auto-refresh mode and 2.47 mA (± 0.01 mA) for one-shot mode. It would correspond to an increase in the lifetime from approximately 12 days to more than 18 days with a 700 mAh battery.

3.2 Detecting the One-Shot Mode on TL Traces

The type of situation we want to detect involves software intended to read a fresh value of the temperature at a regular rate. We assume the one-shot mode is always less efficient than the auto-refresh mode when used with a period $P_r < 1$s. P_{max} and P_{min} are the respective maximum and minimum possible auto-refresh periods of the sensor (here $P_{max} = 1$s and $P_{min} = 13$ms). The decision is given by Table 1. The last line shows a hypothetical case where the software needs a new value more than once every 13 ms. It might be useful for some critical applications where the temperature is rising up extremely fast, but it cannot be accomplished in auto-refresh mode, the one-shot mode should be used. In this case this is not a matter of power efficiency so this is out of our scope.

We can observe the accesses to the sensor registers: reading the value, and requesting the one-shot mode. The tricky part is to measure time between requests to estimate the period, because we work with a simulated model of the hardware platform for which timing is always an approximation; moreover, even with a simple loop code, the software on the real platform does not read the value on a strictly periodic way.

We measure the period several times and compute the average.

Table 1: Detecting Inefficient Uses of the Sensor

	One-shot	Auto-Refresh
$P_{max} > P_r > P_{min}$	inefficient (too many transactions)	efficient
$P_r > P_{max}$	efficient	inefficient (some values will be overwritten before they are read)
$P_r < P_{min}$	correct	wrong (some values will be read twice)

4 IMPLEMENTATION AND EVALUATION

4.1 The TL Platform

The virtual prototype is implemented as a SystemC TL model of the STM32 MCU, including an instruction-set-simulator provided by ARM, and a TL model of the LPS22HB temperature sensor. For the first example (polling loops) we observe bus transactions. Direct Memory Interface (DMI) has to be disabled to ensure transactions visibility. For the second example, we observe the transactions on some registers of the sensor: `CTRL_REG2` is written when a value is asked in one-shot mode, `TEMP_OUT_H` and `TEMP_OUT_L` contain the temperature value, `CTRL_REG1` controls the auto-refresh rate.

4.2 How to Add Monitors

In our proof-of-concept implementation, the monitors are first implemented as classes in Python scripts. The scripts are called at the beginning of the SystemC simulation, and instantiate Python objects representing the monitors. Using Python allows high flexibility and coding simplicity: adding a new monitor can be done without heavy C++ re-compiling of the platform.

In the TL model, the bus and the sensor registers are SystemC modules that receive transactions. The object constructor of the monitors sets a watchpoint on a SystemC module and is programmed to trigger when the module receives a transaction with certain conditions, which may involve the metadata of the transaction. When it happens, the SystemC simulation is paused, the state of the hardware model can be inspected (for instance bits in control registers), the simulation date and all the metadata of the transaction are given to a Python method of the monitor. When the method returns it gives control back to the SystemC simulation.

The monitors can wait for a user action before giving control back to the SystemC simulator. This can be useful for the user to see the current instruction of the embedded software, via a connected debugger, or to inspect the internal state of the monitors.

4.3 Monitor for the Detection of Polling Loops

Observing the transactions on the bus is always available in TL models, but not necessarily the values of the registers internal to the CPU, like the program counter, if the CPU model is vendor-specific. So our polling loop detection works only with the memory accesses and the accesses to registers in components distant from the CPU. The monitor sets a watchpoint on the bus that triggers on the condition READ. When SystemC gives control to the Python script, it also transmits the target module name and the address.

4.3.1 Filtering Bus Transactions. We need to observe the sequence S of read addresses issued on the bus by running the polling loop alone. But other accesses are due to the execution of the interrupt handlers on the same CPU (or even other threads if some scheduler was used, but we consider bare-metal implementations). It is possible to ignore any transaction issued while an interrupt is in *active* state. The virtual prototype includes a model of the nested vector interrupt controller (NVIC) that provides a register to indicate whether any interrupt is active or not. An interrupt is active when the CPU is running the interrupt handler (see STM32 reference manual [5]). As the first monitor is disabled when an interrupt is active, we instantiate another monitor — destroyed at the end of the interrupt handler — in order to detect polling loops inside it. So the maximum number of monitor instances is the number of possible interrupts: 255. Transactions due to fetching instructions from memory can also be filtered out: the memory location of the code is known so the corresponding addresses can be ignored.

4.3.2 Recognizing a Polling Loop Sequence. We now work only with READ transactions observed on the bus, targeting the data memory section and the peripherals. As mentioned in section 2.4, the monitor looks for the regular expression $\mathtt{x}^{[n,+\infty]}$ | $(\mathtt{xy})^{[n,+\infty]}$ | $(\mathtt{xyz})^{[n,+\infty]}$ in the sequence of transactions. The recognizer is fully implemented in Python in the monitor class, initialized with a given maximal size of the repetitive patterns to be recognized P, and a minimum number of iterations n. We use $P = 3$ and $n = 21$ in the example.

4.4 Monitor for Sensor Mode Advice

We consider that the user provides our tool with the reference of the sensor used (here LPS22HB); the tool can instantiate a corresponding TL model enriched with the detection of the available refresh periods.

The monitor sets three watchpoints: (1) On the one-shot bit of the register `CTRL_REG2` triggering on the condition WRITE, (2) on the register `TEMP_OUT_L` triggering on the condition READ and (3) on the register `TEMP_OUT_H` with the same condition. When a watchpoint is triggered, the monitor updates the associated measured periods with the simulation date. It then checks for the inequalities of the table 1 and warns the user in the inefficient cases. The monitor also checks the field `ODR` of the register `CTRL_REG1` that indicates the sensor mode (`ODR` $\neq 0$ for auto-refresh, 0 for one-shot).

4.5 Example Results as Shown to the Developer

In our example, Fig. 6 is the message displayed when the monitor detects an inefficient use case of the sensor. If a debugger is connected then the simulation is stopped and it indicates that the software is currently running the `TEMP_SENS_Set_One_Shot()` function in the call stack. Fig. 7 is the message for a polling loop. The debugger indicates that the software is currently running `HAL_Delay()`.

```
Possible inefficient use of the sensor:
top.NODE_0.LPS22HB.registers.CTRL_REG2
ONE-SHOT asked with a period: 803,135,151 ns.
auto-refresh at 100 ms might save some transactions.
```

Figure 6: Detection of Inefficient Uses of a Sensor

```
Polling suspected at addresses 0x20000260, 0x20017FD0
target: top.NODE_0.NUCLEO.STM32.RAM
You might consider putting the CPU in sleep mode and
programming a wake-up interrupt!
```

Figure 7: Detection of a Polling Loop

4.6 Accuracy of the Detection Principle

The `Hal_Delay()` polling loop (Fig. 1) is detected correctly after 21 iterations and the message of Fig. 7 is displayed. Our proposed modification using the RTC wakeup timer, running the harmless loop (Fig 3) is filtered out correctly with the same monitor. Other polling loops are also successfully detected like the example of Fig. 8, although they include some code inside the loop which generates other transactions on the bus.

```
1  /* Wait until ADDR or AF flag are set */
2  tmp1 = __HAL_I2C_GET_FLAG(hi2c, I2C_FLAG_ADDR);
3  tmp2 = __HAL_I2C_GET_FLAG(hi2c, I2C_FLAG_AF);
4  tmp3 = hi2c->State;
5  while((tmp1 == RESET) &&
6     (tmp2 == RESET) &&
7     (tmp3 != HAL_I2C_STATE_TIMEOUT)){
8       if((Timeout == 0) ||
9         ((HAL_GetTick() - tickstart ) > Timeout))
10          hi2c->State = HAL_I2C_STATE_TIMEOUT;
11      tmp1 = __HAL_I2C_GET_FLAG(hi2c, I2C_FLAG_ADDR);
12      tmp2 = __HAL_I2C_GET_FLAG(hi2c, I2C_FLAG_AF);
13      tmp3 = hi2c->State;}
```

Figure 8: A Complex Polling-Loop Example.

4.7 Impact of the Instrumentation on Simulation Time

In all cases above, the embedded software has been compiled with the `-O0` option using embedded GCC. The additional cost of the instrumentation is due to: (1) the context switches between SystemC and Python; (2) the fact that we have to disable the Direct-Memory-Interface (DMI) optimization to ensure visibility of transactions; (3) the cost of the pattern detection algorithms themselves. The cost of detecting sensor uses is negligible, because the registers are accessed very seldom in the main loop, and checking the sensor mode does not require any costly operation.

We therefore focus on the polling-loop monitor. We measured the simulation duration from the start until the first call to `HAL_Delay(800)` in the main function (Fig. 2) with various combinations of DMI and monitor enabling and disabling. In table 2, *None* indicates that the monitor is disabled. *C++* indicates that the

Monitor		*None*	*C++*	*Empty*	*Present*
DMI	*On*	0.7 sec	0.73 sec	2.82 sec	3.06 sec
	Off	1.19 sec	1.24 sec	24.05 sec	28.41 sec

Table 2: Impact of enabling/disabling DMI and monitor during simulation, measured until first call to `HAL_Delay()`.

monitor is implemented directly in the model in C++, *Empty* indicates a monitor that triggers a SystemC to Python context switch at each READ bus transaction, but does nothing. This is useful to evaluate separately the cost of the pattern detection algorithm. The DMI is disabled (*Off*) only on the data section memory zone, hence code fetching remains invisible.

The duration have been measured using the "`time`" linux command, adding the user and the system times. Results show a serious performance breakdown due to Python/SystemC context switches, but this can be avoided writing the monitor in C++. The pattern recognition algorithm itself and disabling the DMI also slow down the simulation. In addition leaving DMI enabled, and instantiating monitors, also have a small impact while it should not (DMI enabling hides all memory bus transactions, so the watchpoint never triggers). This is due to the loading of the python monitor when simulation starts.

These measures show that we can focus on the detection algorithms alone, the instrumentation mechanism being sufficiently efficient.

5 RELATED WORK

[12] proposes to use dynamic binary instrumentation (DBI) to detect *excessive dynamic memory allocations*, and argues that it is a software performance anti-pattern very difficult to detect statically, because it relies on timing (it detects short-lived, high-frequency dynamic memory allocations). The general framework is very similar to ours. We use a simulation of the hardware because power consumption depends on the *state* of the hardware platform, and sometimes on timing. We may use instrumentations of an instruction-set simulator, which has the same potential as DBI. [12] also provides an interesting review of other dynamic approaches for other *software performance anti-patterns*.

[9] is able to detect certain software performance anti-patterns (resource leaks such as CPU, memory, battery) in android applications, which are sometimes imposed by the underlying frameworks. In our case, this would be due to the HAL. The approach is based on analysing the code of the application, thus being static but without the need for the source code. The 8 patterns detected are all related to the static structure of the code.

[10] applies data mining techniques to the analysis of real-time streams in multimedia applications. It helps understand the violations of QoS properties, due to tasks missing their deadline. The implementation is not meant for monitoring contexts, and may need the full trace.

[7] presents efficient algorithms to find *frequent* sequences in databases of ordered transactions. The type of patterns that may be searched resembles what we need for the polling case, but the search criteria is a quantitative measure of the frequency in the whole database. On the contrary, our definition of a polling is local, it does not relate to the frequency, in the whole behavior, of the memory transactions generated by the evaluation of the loop condition. Moreover, the family of algorithms developed for pattern mining in databases does not necessarily work in a *incremental* way, which is necessary for our *monitoring* purpose.

6 CONCLUSION AND FURTHER WORK

We showed on two frequent examples how detecting problems dynamically allows to focus on the real impact of bad software. A piece of code that looks like a polling loop is not always one, and even if it is, it is not always bad for energy consumption. The replacement of the polling loop involves the use of a CPU sleep mode, whose effect cannot be captured at source level. For the sensor example, the dynamic detection on a timed simulation model allows to reason about periods, which, again, would be difficult statically. Further work will be devoted to other classes of power-related problems. Ongoing work is devoted to the design of a dedicated algorithm for the detection of polling loops, exploiting the fact that the pattern `(x|xy|xyz)` has a very particular shape. We will also investigate whether our patterns can be formalized as properties of the traces written in languages like PSL [6], so that they can then be *compiled* into monitors, using techniques similar to those of [13], for instance.

REFERENCES

[1] [n.d.]. IKS01A2 Extension Board. www.st.com/en/ecosystems/x-nucleo-iks01a2.html.
[2] [n.d.]. LPS22HB Temperature Sensor. www.st.com/en/mems-and-sensors/lps22hb.html.
[3] [n.d.]. Nucleo F411RE Board. www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-mpu-eval-tools/stm32-mcu-mpu-eval-tools/stm32-nucleo-boards/nucleo-f411re.html.
[4] [n.d.]. STM32CubeMX. www.st.com/en/development-tools/stm32cubemx.html.
[5] [n.d.]. STM32F411RE. www.st.com/content/st_com/en/products/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus/stm32-high-performance-mcus/stm32f4-series/stm32f411/stm32f411re.html.
[6] Ben Cohen, Srinivasan Venkataramanan, and Ajeetha Kumari. 2004. *Using PSL/Sugar for formal and dynamic verification: Guide to Property Specification Language for Assertion-based Verification.* VhdlCohen Publishing.
[7] Minos N Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 1999. SPIRIT: Sequential pattern mining with regular expression constraints. In *VLDB*, Vol. 99. 7–10.
[8] Franck Ghenassia. 2005. *Transaction Level Modeling With SystemC: TLM Concepts And Applications for Embedded Systems.* Springer-Verlag.
[9] Geoffrey Hecht, Romain Rouvoy, Naouel Moha, and Laurence Duchien. 2015. Detecting antipatterns in android apps. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems.* IEEE Press, 148–149.
[10] Oleg Iegorov, Vincent Leroy, Alexandre Termier, Jean-François Méhaut, and Miguel Santana. 2015. Data Mining Approach to Temporal Debugging of Embedded Streaming Applications. In *Proceedings of the 12th International Conference on Embedded Software (EMSOFT '15).* IEEE Press, Piscataway, NJ, USA, 167–176. http://dl.acm.org/citation.cfm?id=2830865.2830884
[11] Philippe Magarshack. 2018. Accelerating IoT Device Development — from Silicon to Developer Tools, Keynote Speech. In *DVCon Europe.*
[12] Manjula Peiris and James H Hill. 2016. Automatically Detecting Excessive Dynamic Memory Allocations Software Performance Anti-Pattern. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering.* ACM, 237–248.
[13] Laurence Pierre and Luca Ferro. 2008. A tractable and fast method for monitoring SystemC TLM specifications. *IEEE Trans. Comput.* 57, 10 (2008), 1346–1356.
[14] Connie U Smith and Lloyd G Williams. 2000. Software performance antipatterns.. In *Workshop on Software and Performance*, Vol. 17. Ottawa, Canada, 127–136.