JIT-Based Context-Sensitive Timing Simulation for Efficient Platform Exploration

Alessandro Cornaglia
Md. Shakib Hasan Alexander Viehl
FZI Research Center for Information Technology
Karlsruhe, Germany
{cornaglia, mhasan, viehl}@fzi.de

Oliver Bringmann
Wolfgang Rosenstiel
University of Tübingen
Tübingen, Germany
{bringman, rosenstiel}@uni-tuebingen.de

Abstract— Fast and accurate predictions of a program's execution time are essential during the design space exploration of embedded systems. In this paper, we present a novel approach for efficient context-sensitive timing simulations based on the LLVM IR code representation. Our approach allows evaluating simultaneously multiple hardware platform configurations with only one simulation run. State-of-the-art solutions are improved by speeding up the simulation throughput relying on the fast LLVM IR JIT execution engine. Results show on average over 94% prediction accuracy and a speedup of 200 times compared to interpretive simulations. The simulation performance reaches up to 300 MIPS when one HW configuration is assessed and it grows up to 1 GIPS evaluating four configurations in parallel. Additionally, we show that our approach can be utilized for producing early timing estimations that support the designers in mapping a system to heterogeneous hardware platforms.

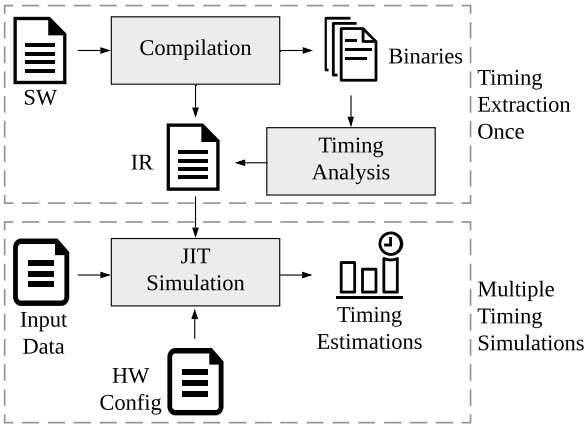

Fig. 1. Fast design space exploration analyzing multiple hardware target platform configurations or software MPSoC partitions.

I. Introduction

Non-functional properties, such as the execution time of a program, are crucial aspects in the embedded systems domain. During the early stages, fast timing estimations are essential. In fact, system designers are often interested in determining the most suitable System-on-Chip (SoC) platform for their system or, in case of heterogeneous Multi-Processor System-on-Chip (MPSoC) architectures, in determining how to partition the application between different processing units. In this particular setting, the selection of the most suitable configuration usually depends on the timing requirements.

Predicting the execution time of a program for a target hardware platform is a hard task. During the last decades, chip manufacturers introduced new complex mechanisms to satisfy the continuous request for higher computation capabilities from the industry. Nowadays, common hardware platforms are equipped with multiple superscalar processors (heterogeneous and multicore CPUs) that include complex mechanisms, such as out-of-order execution, multiple level caches, smart prediction units and others. All these mechanisms attempt to improve the system performance but, at the same time, they reduce the system analyzability. In fact, they appear too complex to be modeled for timing analysis and, in most cases, their documentation is not completely available due to intellectual property restrictions. For this reason, measurement-based timing analysis approaches are preferable to static analysis because they are based on mere timing observations instead of in-depth modeling of complex hardware mechanisms (which is time-consuming and error prone).

Several analysis approaches try to address these intricate issues. One of the preferred techniques for timing estimations relies on system simulations. The different simulators can be characterized by two main quantitative metrics: accuracy and throughput. Among the simulation approaches, context-sensitive simulations proved to be an attractive approach generating rapid and accurate estimations. Context-sensitive simulations also allow evaluating multiple program input data and therefore considering run-time software variability.

In this paper, we present a new measurement-based methodology that allows executing fast and accurate context-sensitive simulations for evaluating the execution time of software programs on complex hardware platforms. The simulations execute LLVM IR code, the compiler's internal hardware independent intermediate representation. As shown in Fig. 1, the approach requires running the timing extraction phase for information extraction only once for each hardware platform configuration. Thereafter, multiple context-sensitive LLVM IR simulations can be executed considering different input data sets and hardware platform configurations. Our methodology enriches the state-of-the-art by: 1) Improving the simulation throughput utilizing the fast LLVM Just In Time (JIT) execution engine. 2) Defining a fully automatic and exact LLVM IR to binary control flow graph mapping. These properties enable our novel approach to offer a further evaluation capability. High simulation throughput and accurate mapping enable to execute fast simulations for producing early timing estimations for the design space exploration of heterogeneous systems considering complex MPSoCs.

The remainder of the paper is organized as follows: an overview of related work is presented in Section II. Thereafter, Section III describes the proposed approach workflow and Section IV presents its evaluation. Section V introduces the MPSoC analysis extension and finally, Section VI concludes the paper with summary and future work.

978-1-7281-4123-7/20/$31.00

II. Related Work

Multiple factors complicate the timing analysis of programs considering different hardware platforms. In fact, the execution time of a program on a given target platform does not depend only on the hardware characteristics of a system. Timing estimation techniques need to consider as well all the optimizations applied at compile-time and the specific program input data. Compiler optimizations can substantially change the high-level structure of a program depending on their objectives. Also, varying the input data may determine, at run-time, different control paths on the control flow graph causing different timing behavior for the same executable.

In this setting, context-sensitive timing simulations have proved to be a valid timing analysis technique for fast design-phase exploration. The different timing estimations produced by these techniques ensure a high level of accuracy, even when analyzing complex hardware target platforms. The literature reports that context-sensitive timing simulations are control flow driven simulations that have been applied considering multiple program abstraction levels. The highest abstraction level for these simulations relies on the program's source code as shown in [1], [2], [3]. Source-level timing simulations generally require two steps before simulating. Initially, timing information is extracted for different parts of a program and consequently, the information is annotated directly in the source code. During the simulation, the produced annotation allows considering target timing information depending on the visited path in the control flow graph (CFG). In contrast, the lowest simulation abstraction level relies on the target binary representation of a program [4], [5], [6]. In this case, fast simulations are executed by directly running the target binary code on emulators on a fast host machine. During the simulation, the emulator updates the timing estimation depending on the visited control flow and fetches the associated timing data from an external timing database. Finally, a third level of simulation abstraction is based on the source code intermediate representation (IR) of a given compiler. An example has been presented in [7], where the authors rely on the LLVM Compiler Infrastructure [8]. Similarly to binary simulations, context-sensitive IR simulations need an IR execution engine that determines the program flow path and that consequently updates the timing estimation.

Binary-level context sensitive simulations can suffer from undesired slow-down introduced by the necessary instruction-set architecture (ISA) translation performed by the system emulator [9]. This type of approaches fully relies on the emulator support. Additionally, every simulation is limited to a specific hardware platform configuration and to a predefined set of compiler optimizations. Source-level and IR-level simulations do not require ISA translations and, for this reason, they may offer higher throughput. Source-level simulations directly execute binary code compiled for a host machine, while IR-level simulations execute the IR code in a sort of virtual machine. Inevitably, both the approaches in these categories require a mapping between the code binary representation and the one that is utilized during the simulations. The mapping is necessary reasons of accuracy. In fact, the timing information is always extracted at the binary level via measurements. In contrast, simulations are executed on a program abstraction layer that may be based on a different CFG structure.

It is commonly easier to generate an IR to binary CFG map than a source code to binary one. In fact, the source code does not consider compiler optimizations at all. In contrast, the IR code structure is the effect of the front end compilation process and, after the middle end phase (or optimizer), it already includes all the requested hardware independent optimizations. Consequently, the IR structure is closer to the binary one and easier to map. In [1] a complex source to binary mapping is presented. The map is generated relying on the debug information generated during the compilation. Unfortunately, it is not always possible to rely on this information. The compiler has to support it and, in some circumstances, it may be inaccurate due to complex compiler optimizations (mainly performed during the front end and middle end compilation activities). In contrast, multiple methodologies [10], [11], [12] show that easier algorithms, independent from the compiler support, can be utilized to produce IR to binary mapping.

The new methodology we propose belongs to the IR context-sensitive simulations category and it is inspired by the simulation approach previously presented in [7]. The authors implemented SIMULTime, a timing simulator based on the interpretive LLVM IR execution engine for evaluating in parallel multiple SoCs and compiler optimizations configurations. In contrast to this approach, we enrich the state-of-the-art by proposing LLVM IR timing simulations based on the faster JIT execution engine. In addition, we propose an improvement for the IR to binary CFG mapping algorithm described in [10] by making the algorithm fully automatic. With our improvement, the algorithm produces exact mappings solving possible mapping ambiguities without expert supervision. High simulation throughput and accurate mappings enable our methodology to execute fast timing simulations and produce timing estimations for the design space exploration of heterogeneous systems considering complex MPSoC platforms.

III. Simulation Methodology

In this section, we initially present the fundamental building blocks that compose the proposed LLVM IR context-sensitive timing simulation methodology. Thereafter, we present the simulation framework workflow, shown in Fig. 2, that allows evaluating multiple SoC configurations.

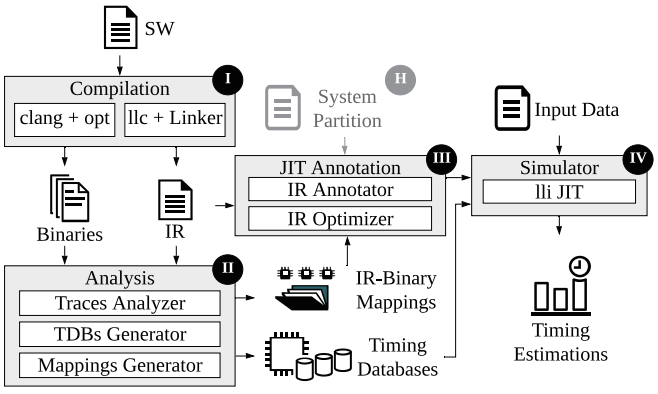

Fig. 2. Multiple JIT-based simulations can be executed considering different input data and reusing TDBs and IR-binary mappings previously generated.

A. Implicit Hardware Modeling

The proposed methodology belongs to the measurement-based timing analysis approaches. Measurements can either be extracted directly by running an application on a target platform or, if the hardware is not available, on an accurate simulator. Complex programs can be stimulated differently with multiple input data sets. Due to the complexity of programs and the wide range of possible input data, it is not feasible to measure every possible path in the CFG and consider every input data value. Under the assumption that programs spend most of their time in loops and recurring functions, we identified the $VIVU(n,k)$ (virtual inlining and virtual unrolling) context mapping function [13] as an appropriate method for implicitly modeling complex hardware platforms and their mechanisms via measurements. Accurate timing descriptions of loops and blocks of code can be expressed considering complex stateful hardware resources such as cache memories, pipelines, branch predictors, and others.

This mapping relies on the analysis of an interprocedural CFG that describes the structure of a complete program. Execution paths on this graph can be expressed via call strings. A call string contains the concatenation of all the labels of the call edges (traversed edges while visiting the basic blocks). A context in the program is described by a sequence of pairs consisting of a non-recursive call edge and the recursion count. A call string may theoretically be infinite due to recursion or unbounded loops in the program and, consequently, it may be necessary to determine an infinite number of contexts. For this reason, the $VIVU(n,k)$ mapping function utilizes two parameters to reduce this number. The first parameter n limits the maximum recursion count to consider. The second parameter k limits the number of elements in a context.

For a specific binary executable, it is possible to generate multiple timing databases (TDB) arranging the extracted measurements according to the VIVU mapping function. A TDB contains timing information for a specific hardware target configuration and for a specific set of compiler optimizations. A selected simulator can query at run-time one or multiple TDBs and consequently update the timing predictions. Both simulation performance and accuracy are influenced by the configuration of the n and k parameters. High values for both parameters ensure high accuracy but limited simulation speed and, vice versa, low values improve the speed by sacrificing the accuracy. Depending on the simulation objective, it is necessary to determine a tradeoff between them.

B. IR to Binary CFG Mapping

Our methodology requires an IR to binary CFG mapping mechanism. This is a common requirement for approaches that are not directly based on the software binary representation. Errors or incomplete mapping results can lead to a reduced prediction accuracy. From the large number of mapping approaches available in the literature, we decided to utilize for this delicate task the mapping algorithm presented in [10]. This algorithm allows producing exact mappings relying on two simple numerical metrics that are flow value and nesting level. Unfortunately, the algorithm is not fully automatic. In fact, it can fail in case the graphs contain ambiguities. An ambiguity arises when nodes in similar positions have the same metrics values. Fig. 3 shows an ambiguity example problem, where nodes in similar positions have the same flow value (F) and nesting level (N). The solution provided by the authors in [10] requires the support of an expert to disambiguate these cases. They expect that an expert solves the ambiguities by inspecting and consequently comparing the code contained in both the IR and binary blocks. This can be a hard and expensive task, especially in case of large applications. In this regard, we conducted some experiments and we observed that mismatches introduced by an expert can reduce the estimations accuracy by 40%.

Considering the importance of this task, we propose an extension to the algorithm that makes it fully automatic. The extension allows producing exact mappings even in case of complex applications. We propose to replace the expert knowledge with information extracted from both binary and IR traces. Traces contain visited control flow paths during the execution of a program. Therefore, we decided to utilize binary and IR traces generated with the same program input data in the disambiguation task. The TDB generation procedure requires extracting multiple binary traces and they can be re-utilized. IR traces containing an ordered list of visited basic block labels can be quickly generated by JIT executing properly instrumented IR code. Generating the necessary IR traces introduces a minimal overhead in the complete methodology but it is fundamental in ensuring high accuracy.

The fully automatic mapping process requires two consecutive phases. Initially, the original mapping algorithm is executed and eventual ambiguities are identified. During the second phase, which is executed only in case of ambiguities, the information extracted from the traces is utilized for adjusting incomplete mappings. Fig. 3 shows an example of mapping adjustment. In the first phase the original algorithm correctly succeeds in mapping the IR basic blocks I_k and I_z but it fails for the other two blocks because of an ambiguity (graphically represented by a question mark). Therefore, in the second phase, the ambiguity is solved relying on path information extracted from traces (gray path). Initially the algorithm decides to associate I_x with B_x and consequently it completely solves the ambiguity associating the remaining node I_y with B_y.

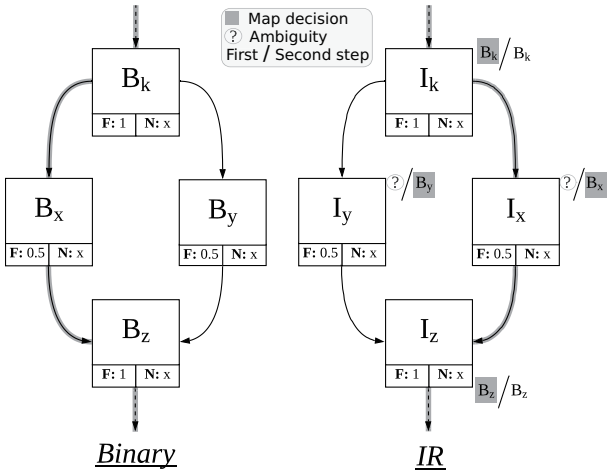

Fig. 3. IR to binary mapping ambiguity problem and suggested trace-based solution: IR and binary traces are automatically compared to solve possible initial algorithm ambiguities.

C. IR Simulation Code Annotation

The LLVM IR code is organized in compilation modules that collect program information. Modules describe functions and global variables that are contained in a program. Each function, in turn, contains at least an entry basic block followed by potential ones. As shown in the left part of Fig. 4, every basic block is identified by a label. Each basic block contains, in a strict order, zero or multiple Phi-instructions and zero or multiple instructions followed by a terminator instruction. The IR instructions are human-readable and they are expressed in the static single assignment form (SSA).

The LLVM IR syntax does not consider a function call instruction as a basic block terminator. In contrast, the $VIVU$ mapping theory considers these instructions as initiators of edges in the context call strings. Consequently, during the TDB generation, where an interprocedural CFG is considered, every function call instruction represents a basic block terminator. In order to ensure high-level timing estimations accuracy, it is important to consider this discrepancy while running context-sensitive timing simulations.

Our approach allows executing fast JIT-based context-sensitive timing simulations by executing annotated LLVM IR code. We only annotate the IR code used during the simulations. We do not annotate the optimized IR code that is provided in input for the back end compilation. The right side of Fig. 4 shows how we propose to annotate the simulation code considering the terminator instructions discrepancy problem. The annotation is performed by running an additional optimizer pass we created on the previously generated optimized IR. The annotation consists of an IR function call that, at run-time, forces the IR execution engine to update the timing prediction values by querying the necessary timing databases. The function is implemented in an IR library linked to the simulation code. The annotation can be inserted in two possible locations: at the beginning of the basic block instructions immediately after the possible Phi-instructions (if any), and after every function call instruction. The former location helps managing the case where a basic block is entered, the latter manages instead the case where a basic block execution is resumed after a function call returns.

D. Workflow

The proposed methodology workflow requires four distinct interconnected phases. The phases are shown in Fig. 2 and they are identified by circled roman numerals. The workflow starts with the compilation phase Ⓘ, which has to be executed only once. The front end and the middle end (respectively called *clang* and *opt* in LLVM) generate the IR code. The back end (*llc*) and linker terminate the cross-compilation producing one binary executable for target configuration (target platform, hardware dependent optimizations and memory mapping). Consequently, the analysis phase Ⓘⓘ can be started to produce essential information for the timing simulations. The analysis has to be conducted for all the hardware target configurations but only once per program. Its objective is to generate all the necessary TDBs and the associated IR to binary CFG mappings in three main activities. Initially, timing traces are extracted directly from the hardware platforms. One TDB per hardware target configuration is generated analyzing the traces and applying the previously described $VIVU$ function mapping. In parallel, the necessary exact IR to binary CFG mappings can be generated. Both the program representations are analyzed and consequently the previously described extended mapping algorithm is executed. The third phase Ⓘⓘⓘ starts with the annotation of the IR code. Relying on the IR-Binary mapping files generated during the previous phase, an additional *opt* pass allows inserting the necessary instrumentation in the IR file designed for the simulation. This operation is specific for a set of configurations and it has to be repeated every time the set changes. Once the annotated IR has been generated, we run additional passes to further optimize the IR simulation code. These passes improve the simulation performance considering the host machine architecture where the simulations will be executed.

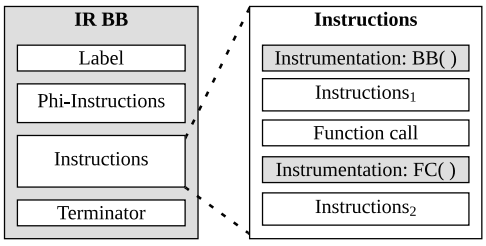

Fig. 4. IR simulation code annotation: Annotation function calls are inserted at the beginning of the BBs and after every function call instruction.

At the end of the third phase, all the necessary information for running JIT-based contex-sensitive simulation has been generated. The timing simulations Ⓘⓥ are executed via *lli*, an LLVM tool that directly executes IR code. The simulations can start selecting the *lli* JIT execution engine and providing in input both the necessary generated TDBs and the annotated IR code. During its execution, the simulator updates the timing predictions by fetching relative time values from the TDBs according to the IR annotation. Multiple fast simulations can be executed considering different input data.

IV. JIT-Based Simulation Evaluation

A first objective of our evaluation is determining the simulation throughput speedup that JIT-based simulations provide compared to the interpretive ones. We expected a substantial speedup maintaining the high-level of accuracy intrinsic in context-sensitive simulations. For this purpose, we initially cross-compiled the Mälardalen benchmarks [14] for the ARM Cortex-A15 processor included in the TI EVM-K2E hardware platform [15]. Thereafter, timing traces have been extracted directly from the target via the non-intrusive TRACE32 tracer [16]. Two timing databases per benchmark, $VIVU(20, 20)$ and $VIVU(\infty, \infty)$, have been generated following the procedure previously described. Thereafter, the different simulation throughputs resulting from simulations based on the interpreter execution engine and the faster JIT have been compared. These results are shown in Fig. 5. The speedup achieved considering the two different timing databases configurations confirms the speed benefit of running JIT-based simulations instead of interpretive simulations. Our measurements show an average speedup of 73 for TDB $VIVU(20, 20)$ configuration and 45 for TDB $VIVU(\infty, \infty)$, while the accuracy is unaltered (error percentage lower than 6%). The maximum simulation

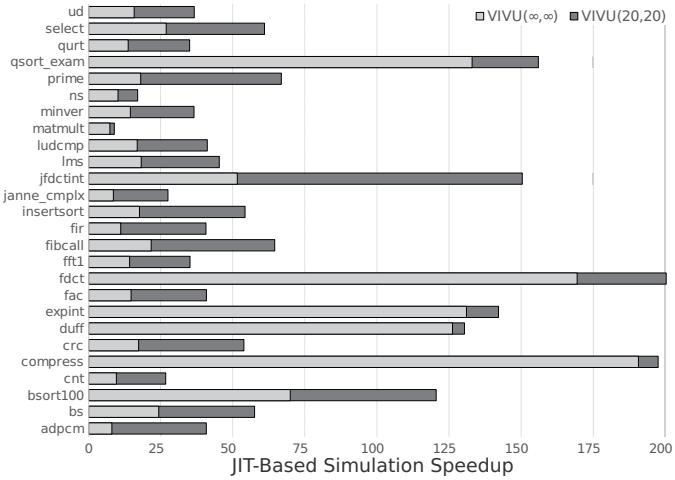

Fig. 5. Observed single SoC simulation throughput speedup comparing JIT and interpretive simulations.

throughput value has been observed for the `fdct` benchmark fetching timing information from the TDB $VIVU(20,20)$. In this specific case, the JIT-based simulation achieved a throughput value of 302 MIPS (millions simulated ARM instructions per second) on a normal host machine with an Intel Core i5 CPU 750 at 2.2 GHz.

A second objective of our evaluation is assessing the benefit of evaluating multiple hardware platforms in parallel by executing only one JIT-based simulation. An improvement in the simulation throughput represents an essential prerequisite for the later presented MPSoC simulations. For this reason, we decided to evaluate the simulation throughput considering four complex SoCs that include four distinct ARM processors: TI EVM-K2E [15], TI Hercules RM57LHDK [17], Xilinx ZedBoard Zynq-7000 [18] and Hitex LPC4350 [19]. Respectively, the included ARM processors are: Cortex-A15, Cortex-R5, Cortex-A9 and Cortex-M4. Simulating the Mälardalen benchmarks with different data sets, we have been able to determine for each of them the most performant SoC. The results for six significant benchmarks considering only TDBs $VIVU(20,20)$, are plotted in Fig. 6. For all of them, we observed that increasing the number of configurations to analyze in parallel implies an increment in the simulation throughput (for both $VIVU$ configurations). The highest reached simulation speed has been observed evaluating in parallel four hardware SoC for the `fdct` benchmark. In this case, the maximum simulation speed has been 1.13 GIPS. On average,

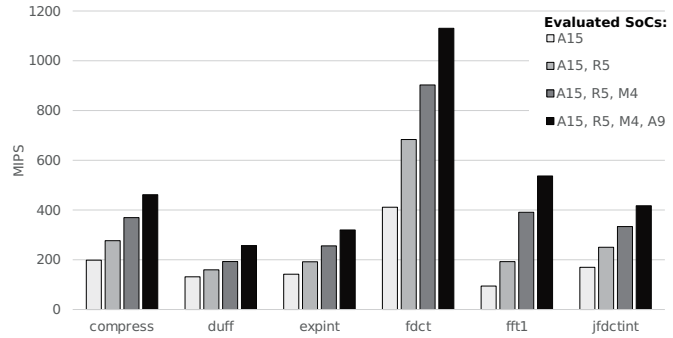

Fig. 6. Simulation throughput increase in case multiple configurations are evaluated in parallel in one simulation.

simulating four configurations in parallel showed a simulation speed of approximately 210 MIPS which represent a speedup of 140% compared to single configuration simulations.

V. MPSoC Early Timing Estimations

The results collected during the evaluation of the JIT-based simulations show that the proposed methodology ensures significant simulation throughput and highly accurate results. The simulation throughput is preserved even when multiple SoCs are evaluated in parallel. Therefore, in addition to the previously described workflow, we show that the proposed methodology can be utilized during the early design space exploration activities, producing early estimations for the execution time of heterogeneous applications considering different MPSoC mappings. Below, we first show how to generate MPSoC timing estimations and then exemplary results.

A. Workflow Extension

In order to produce timing estimations for heterogeneous systems, an additional activity is required in the simulation workflow. This step is represented in Fig. 2 (H) and identified by gray shapes. The basic idea for producing MPSoC timing estimations is to execute LLVM IR context sensitive simulations considering multiple TDBs, one per hardware/software partition. Depending on the partition schema, the JIT-based simulation selects the appropriate TDB to query for updating the timing estimation. Currently, our methodology can analyze synchronous heterogeneous systems and it neglects the synchronization time between the processors. An analyzable heterogeneous system is composed of functional units (FU), collections of functions, that can be mapped to different processors included in an MPSoC.

The additional activity consists in a new annotation pass. This pass annotates the simulation IR code forcing at run-time the JIT execution engine to visit a specific TDB depending on an input configuration. The configuration specifies the heterogeneous partitions and the FUs. Each FU has to be assigned to one of the processors in an MPSoC. The configuration can be generated either by hand or from a model-driven development tool as MATLAB/Simulink in a similar way the authors showed in [20]. In the second case, the model has to be annotated and the auto-generation source code process has to produce one function for each component in the model.

B. Exemplary Evaluation

We evaluated the MPSoC analysis extension with a practical example. We considered a system composed of four interconnected ARM Cortex processors: A9, A15, M4 and R5. We created a software application combining different Mälaralen benchmarks listed in in Table I. The selected benchmarks differ in characteristics and source lines of code (SLOC). The table also describes the software application and its partition. The FUs execution order is: $FU_1 \rightarrow FU_2 \rightarrow FU_3 \rightarrow FU_4$. The complete application has been compiled for all four the processors and the workflow has been followed generating the IR simulation code. We propose two different ways for producing the TDBs to consider during the simulations. A TDB can be generated from traces extracted from the complete execution of a program or only from traces belonging to

TABLE I
HETEROGENEOUS APPLICATION STRUCTURE AND SYSTEM PARTITIONS

Functional Unit	SoC	Processor	Benchmarks	SLOC
FU_1	TI EVM-K2E	Cortex-A15	bsort100	128
			cnt	267
			crc	128
FU_2	TI RM57LHDK	Cortex-R5	fac	27
			fdct	239
			fft1	219
FU_3	Hitex LPC4350	Cortex-M4	fibcall	72
			insertsort	92
			janne_complex	64
FU_4	Xilinx Zynq-7000	Cortex-A9	matmult	163
			minver	201
			ns	535

TABLE II
AVERAGE HETEROGENEOUS SIMULATION VALUES AND PREDICTIONS

Tracing Approach	VIVU	Simulation Time (ms)	MIPS	% Error
Complete Trace	$(20, 20)$	397.931	69.98	26.71
	(∞, ∞)	585.88	47.60	26.91
Trace Partitions	$(20, 20)$	403.852	68.96	8.89
	(∞, ∞)	591.72	47.07	9.31

the appropriate partition. The first possibility results easier, simple end-to-end tracing, but it may entail inaccuracy due to inappropriate timing values for some contexts. The second one is expected to be more accurate but time consuming in the tracing procedure.

The results of our evaluation are collected in Table II. Four different TDBs have been created: we extracted the necessary timing traces considering both the techniques previously described and, for each tracing technique, we generated two TDBs setting the $VIVU$ parameters to (∞, ∞) and $(20, 20)$. Executing the simulations on a normal host machine, their execution completed always in less than one second. When the simpler tracing method is utilized, the percentage of error for the MPSoC timing estimation is around 26%. The error drastically drops to 9% in case the traces are extracted considering the system partitions.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a new approach designed for supporting the early design space exploration activities in the development of embedded systems. Fast and accurate timing estimations of the execution time of a program considering multiple complex hardware target platforms can be assessed by only one simulation. These timing estimations are produced by executing LLVM IR context-sensitive simulations. The simulations rely on the fast JIT execution engine and an on exact IR to binary CFG mappings. The timing estimations are highly accurate, with an error percentage lower than 6%, and the simulation throughput, up to 302 MIPS, is preserved even in the case of multiple SoCs being evaluated in parallel. These two properties allow producing early timing estimations and exploration of different software application's mappings to complex heterogeneous MPSoC platforms.

Prospectively, we intend to improve the MPSoC analysis by also considering asynchronous heterogeneous systems. The extension will allow analyzing more complex scenarios where shared stateful hardware resources and scheduling events influence the timing performance of a system.

ACKNOWLEDGEMENTS

This work has been partially supported by the German Federal Ministry of Education and Research (BMBF) in the ITEA3 projects: REVaMP^2 under grant 01|S16042A and COMPACT under grant 01|S17028C.

REFERENCES

[1] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Fast and accurate source-level simulation of software timing considering complex code optimizations," in *Proceedings of the 48th Design Automation Conference*, pp. 486–491, ACM, 2011.

[2] N. Frid, D. Ivošević, and V. Sruk, "Elementary operations: a novel concept for source-level timing estimation," *Automatika*, vol. 60, no. 1, pp. 91–104, 2019.

[3] Z. Zhao, A. Gerstlauer, and L. K. John, "Source-level performance, energy, reliability, power and thermal (perpt) simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 2, pp. 299–312, 2016.

[4] S. Ottlik, J. M. Borrmann, S. Asbach, A. Viehl, W. Rosenstiel, and O. Bringmann, "Trace-based context-sensitive timing simulation considering execution path variations," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 159–165, IEEE, 2016.

[5] S. Ottlik, C. Gerum, A. Viehl, W. Rosenstiel, and O. Bringmann, "Context-sensitive timing automata for fast source level simulation," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 512–517, IEEE, 2017.

[6] R. Plyaskin and A. Herkersdorf, "Context-aware compiled simulation of out-of-order processor behavior based on atomic traces," in *2011 IEEE/IFIP 19th International Conference on VLSI and System-on-Chip*, pp. 386–391, IEEE, 2011.

[7] A. Cornaglia, A. Viehl, O. Bringmann, and W. Rosenstiel, "Simultime: Context-sensitive timing simulation on intermediate code representation for rapid platform explorations," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pp. 526–531, ACM, 2019.

[8] The LLVM Compiler Infrstructure. https://llvm.org, Last time accessed July 2019.

[9] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of gem5 simulator system," in *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pp. 1–7, IEEE, 2012.

[10] S. Chakravarty, Z. Zhao, and A. Gerstlauer, "Automated, retargetable back-annotation for host compiled performance and power modeling," in *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, p. 36, IEEE Press, 2013.

[11] Z. Wang and A. Herkersdorf, "An efficient approach for system-level timing simulation of compiler-optimized embedded software," in *Proceedings of the 46th Annual Design Automation Conference*, pp. 220–225, ACM, 2009.

[12] O. Matoussi and F. Pétrot, "A mapping approach between ir and binary cfgs dealing with aggressive compiler optimizations for performance estimation," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 452–457, IEEE, 2018.

[13] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand, "Analysis of loops," in *International Conference on Compiler Construction*, pp. 80–94, Springer, 1998.

[14] The Mälardalen WCET Benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html, Last time accessed July 2019.

[15] EVM K2E (Evaluation Modules & Boards). http://www.ti.com/devnet/docs/catalog/thirdpartydevtoolfolder.tsp?actionPerformed=productFolder&productId=20300, Last time accessed July 2019.

[16] Lauterbach Development Tools. https://www.lauterbach.com, Last time accessed July 2019.

[17] Hercules RM57Lx Development Kit. http://www.ti.com/tool/tmdxrm57lhdk, Last time accessed July 2019.

[18] ZedBoard Zynq-7000 ARM/FPGA SoC Development Board. https://www.xilinx.com/products/boards-and-kits/1-elhabt.html, Last time accessed July 2019.

[19] Hitex LPC4350 Evaluation board. https://www.nxp.com/support/developer-resources/nxp-designs/hitex-lpc4350-evaluation-board:OM13031, Last time accessed July 2019.

[20] A. Cornaglia, S. Hasan, A. Viehl, O. Bringmann, and W. Rosenstiel, "Modeltime: Fully automated timing exploration of simulink models for embedded processors," in *AmE 2019-Automotive meets Electronics; 10th GMM-Symposium*, pp. 1–6, VDE, 2019.