Accurate LLVM IR to Binary CFGs Mapping for Simulation of Optimized Embedded Software

Alessandro Cornaglia[1], Alexander Viehl[1], and Oliver Bringmann[2]

[1] FZI Research Center for Information Technology, Karlsruhe, Germany
{cornaglia|viehl}@fzi.de
[2] University of Tübingen, Tübingen, Germany
oliver.bringmann@uni-tuebingen.de

Abstract. In this paper, we present a new approach for mapping LLVM IR to binary machine code for overcoming the current limitations of host-based simulations of performance-critical embedded software imposed by compiler optimizations. Our novel, fully automated mapping approach even copes with aggressive compiler optimizations without requiring any modification to the compiler or the need of expert supervision. Experimental results show that accurate mappings are produced even when compiling with the highest level of optimization (average error below 2%). The proposed simulation methodology provides a speedup of at least 26 compared to the widely used gem5 simulator.

Keywords: Performance Estimation · Host-Based Simulation · IR to Binary CFGs Mapping · Design Space Exploration.

1 Introduction

The design of embedded systems requires dealing with the complexity of the actual processor architectures, the constant need for reducing the waste of precious resources and the strictness of the non-functional requirements. The performance of a program is generally optimized by applying compiler optimizations that focus on increasing the instruction-level parallelism (ILP) by making the best runtime use of the physical resources available on the processor. In this context, the designers need early feedback about the performance of the software program regarding its execution on a given target platform. The evaluation is usually conducted by simulating the target system on a development host machine.

Traditional simulators like instruction-set simulators (ISS) or gate level RTL simulators often cover too many unnecessary details in too much depth that make them too slow and unsuitable for supporting the design activities. Fast host-based simulation techniques are an alternative for tackling this limitation. Host-based simulators execute a software binary version that differs from the one produced for the target. The target performance can be simulated by compiling the program for the host machine and executing it. The simulation can be executed on different software representations such as the source code, its compiler intermediate representation (IR) or the host binary simulation code.

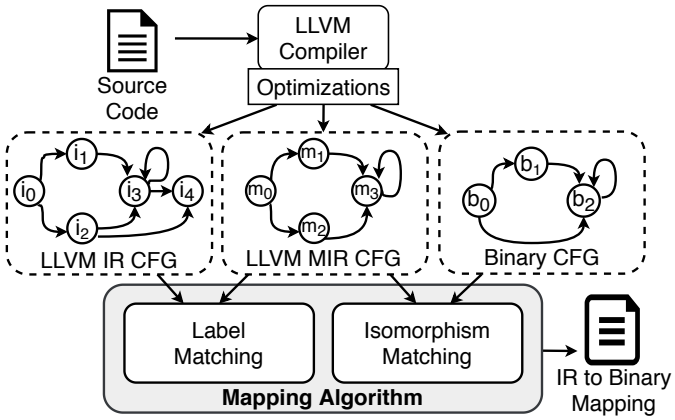


Fig. 1. Proposed two-steps approach for accurately mapping LLVM IR to binary code dealing with the effects of the compiler optimizations.

The simulation speedup provided by the host-based simulation techniques comes at the price of mapping the simulation code representation to the target binary machine code. Unfortunately, this task can be extremely hard and complex because, especially in typical industrial settings, it is desirable to highly optimize the program at compile-time. Aggressive compiler optimizations can substantially change the structure of a program in comparison to the structure of the original source code. In most cases, when the program is highly optimized, these transformations make a direct match between the different representations of the control flow graphs (CFGs) impossible. It is easier to map IR to binary code rather than trying to map the source code directly. The IR is a lower level representation of the source code that is internal to the compiler and it is designed to support architecture-independent compilation passes. Its structure is closer to the binary representation because it already includes the effects of some of the compiler optimizations. Mapping IR to binary machine code only requires considering the effects of the missing back-end transformations.

The main contribution of this paper is the definition of a new and fully automatic approach for generating a precise mapping between LLVM IR and binary machine code that can deal with aggressive compiler optimizations. The methodology relies on the LLVM MIR representation, a machine dependent low-level code representation designed for applying architecture-dependent optimizations. The mapping problem is decomposed by initially mapping LLVM IR to MIR code and consequently mapping the MIR to binary machine code. As a result, the precise mapping allows defining a fast host-based simulation methodology for accurately evaluating non-functional properties of an embedded system, such as timing, performance and power estimations.

The rest of the paper is organized as follows: Section 2 introduces other existing mapping approaches. An introduction to fundamental LLVM concepts is provided in Section 3. The proposed mapping approach is presented in Section 4 and the host-based simulation methodology is described in Section 5. Section 6 shows the experimental results and Section 7 concludes the paper.

2 Related Work

Mapping the structure of the simulation's code representation to the cross-compiled binary code is a common requirement for executing host-based simulations. Unfortunately, this is a hard task, especially in the presence of aggressive compiler optimizations. Some approaches try to directly map the source code to the binary instructions. Others instead rely on the IR code because its structure already includes the effects of the architecture-independent compiler optimizations. The IR structure is consequently closer to the binary in comparison to the structure of the source code. Our mapping relies on a lower level code representation but the methodology is mainly inspired by three different approaches [5, 12, 13] that belong to both of the two categories.

The concept described in [13] proposes an automatic approach for mapping source code statements to the respective binary instructions generated after the program's cross-compilation. This approach relies exclusively on the DWARF debug information generated by the compiler. Unfortunately, even if this information is available, the mapping may turn out to be imprecise or ambiguous due to the effects of aggressive compiler optimizations that break the full traceability between source code and binary instructions [2]. Possible improvements are proposed in [17] and [16], imprecision and ambiguities are reduced by substituting the source code parts subjected to aggressive compiler optimizations with functionally-equivalent optimized IR code. The optimized IR code has a structure closer to the binary representation, helping in generating the mapping but without fully solving the problem even considering hierarchical subgraphs of a CFG as proposed in [11].

An automated flow for mapping IR to binary code is described in [5]. This algorithm implements a heuristic for mapping the basic blocks of the two different program representations relying on their similarities. The similarities are defined considering two numerical metrics: flow value and nesting level. In some cases, the heuristic fails in producing a complete mapping because of ambiguities in the CFGs. If this happens, the mapping process requires the supervision of an expert. The authors in [6] proposed a tracing-based enrichment for making the algorithm fully automatic. The improvement consists in filling the mapping gaps by comparing IR and binary execution traces. On the one hand, the solution can fix the ambiguities problem, but on the other hand, it can be hard to identify the exact input data that leads the execution to visit the desired control flow path.

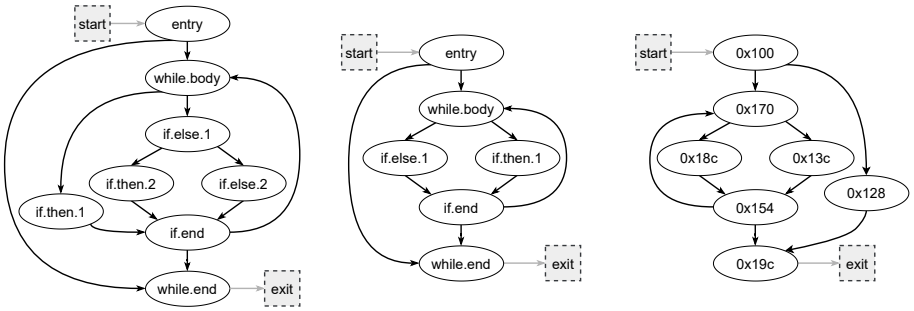
A different approach for mapping IR to binary code with the purpose of performance estimation in native simulation is presented in [12]. By analyzing optimized IR code, this approach can focus on managing only the hardware-dependent optimizations applied during the back-end compilation. In particular, this approach is focused on identifying, characterizing and finally mapping the loops in the two different program representations. The experimental results show that an increase in accuracy can be reached only if all the loop optimizations are correctly identified and managed. However, every single imperfection may introduce a substantial percentage of error.

3 LLVM Background

The LLVM Compiler Infrastructure [9] provides tools for analysis and for both static and dynamic compilation of programs. The compiler's front-end, `clang`, translates the source code to bitcode. The bitcode is the LLVM IR, it is architecture independent and it is structured in modules. The architecture-independent optimization passes performed by `opt`, the middle-end, are applied to this intermediate representation. It is possible to link multiple IR modules together via `llvm-link`. The compilation, or cross-compilation, process is concluded by `llc`, the back-end, that finally translates the bitcode to binary machine code.

Two further architecture-dependent code representations are considered in LLVM: The LLVM Machine Code (MC) [15] and the LLVM Machine IR (MIR) [14]. The former is a representation for an object file that is similar to the binary representation and does not contain any high-level information stored in the bitcode. The latter is an architecture specific translation of the bitcode that is utilized by the back-end for applying architecture-dependent optimizations. The structure of an MIR module resembles the structure of its original IR module. At the same time, the MIR structure is closer to the binary representation because it includes most of the effects caused by the architecture-dependent optimizations, including the ones that change the program structure. For instance, an MIR module already includes the effects of aggressive loop optimizations such as loop unrolling, loop-invariant code motion, loop inversion and others. During the final compilation phases, for every function in the MIR module, the MIR basic blocks are sequentially translated in order into LLVM MC instructions.

The key idea of our novel mapping mechanism consists in also considering the program's structure at the MIR level. Instead of directly map the IR structure to its corresponding binary, we propose to perform an additional step that requires matching the MIR representation to both the IR and binary representations. As shown by the consequent IR, MIR and binary CFG representations of a given function in Fig. 2, this double mapping mechanism allows to reduce the problem complexity even when aggressive architecture-dependent optimizations are applied. The problem's simplification derives from the fact that: 1) There exists an implicit direct connection between the IR and MIR basic blocks' labels and, 2) The MIR structure is very close to the structure of the resulting executable.



(a) Optimized IR CFG. (b) Optimized MIR CFG. (c) Optimized binary CFG.

Fig. 2. Control flow graphs at different code representations for the same function.

4 Mapping Algorithm

The proposed novel mapping approach consists of two separate phases, as shown in Fig. 1. The first one allows the generation of a mapping between IR and MIR CFGs relying on the definition of the label matching algorithm. The second one allows mapping MIR to binary CFGs relying on the isomorphism matching algorithm that automatically generates an isomorphism between them. The combination of these two mappings together allows producing a precise mapping from LLVM IR to binary machine code. The two phases are presented in the next sections with the help of the support of the CFGs provided in Fig. 2. The terms basic block and node will be utilized in an interchangeable way by following this definition: a CFG is a directed graph $CFG = (N, E)$ where every node $n \in N$ corresponds to a basic block and each edge $e = (n_i, n_j) \in E$ identifies a connection between two blocks. The mapping is intended to define a match between paths (sequence of edges) in the CFGs. This granularity allows considering, for every single function in the program, only the effects of the optimizations that change the structure of a graph. The mapping is not intended to model any of the optimizations that can change the internal structure of the basic blocks. This assumption does not limit the possibility of future finer granularity extensions.

4.1 Label Matching

The basic block labels are used to support the automatic mapping of IR to MIR CFGs. The labels are internal to the compiler, specific per function, completely independent from the debug symbol information (that can be unavailable or imprecise) and each of them identifies a specific node in the graph. An example is given in Fig. 2(a) and Fig. 2(b). The two graphs are similar but a direct mapping between the labels of the two representations is not possible because of the effects of potential optimizations. MIR nodes can be removed or inserted [1]. An insertion implies the appearance of a new MIR node identified by a new synthetic label. These labels are not present in the IR CFG. On the opposite, a removal causes the IR CFG to contain a label not present in the MIR CFG.

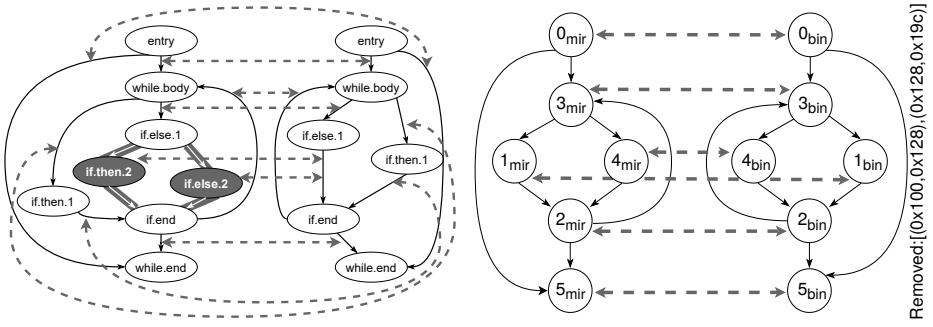
The algorithm starts by labeling the nodes of the CFGs with the labels of the corresponding basic blocks. The labels are consequently utilized for defining an initial partial mapping between IR and MIR edges. Relying on the labels, the algorithm identifies the IR edges that have been preserved from the optimizations and that are part of the graphs of both the representations. The corresponding edges can be directly matched as graphically shown by the dashed arrows in Fig. 3(a). The complete mapping is finally obtained by considering the remaining unmatched edges. Consecutive unmatched edges are grouped in paths. Iteratively, and for both the graphs, the algorithm maps a path $p = ((n_i, n_j),\ \ldots\ , (n_x, n_y))$ of one graph to the shortest path (considering possible back-edges) in the other graph between the nodes identified by the labels of nodes n_i and n_y. These matches, represented as dotted arrows in Fig. 3(a), complete the IR to MIR mapping.

4.2 Isomorphism Matching

The process for defining an MIR to binary mapping requires the generation of an isomorphism between their CFGs. The isomorphism can be obtained by modifying the structure of the graphs of both the representations while preserving their original control flow paths. In fact, similarly to certain compiler optimizations, specific nodes can be removed in order to obtain two graphs with an isomorphic structure. The isomorphism ensures the definition of a unique mapping between the edges and paths of the original graphs as described in Algorithm 1.

The algorithm starts by labeling the nodes of the two graphs with unique IDs. These IDs are essential for the isomorphism-based direct mapping. As discussed in Section 3, the basic blocks translation from MIR to binary machine code reflects the order of their appearance in the MIR module. Therefore, every node in the MIR CFG is labeled with an integer ID relying on the order of appearance in the MIR function. In a similar way, it is possible to label the binary nodes relying on their start address (or offset) reported in the binary file.

The algorithm continues by coloring the graphs for identifying the necessary nodes that have to be removed for obtaining the isomorphism. Initially, all the nodes are colored in white. Consequently, all the nodes that are eligible to be removed are colored in black. A node has to be colored in black if, ignoring potential back-edges, its in-degree is equal to one and its out-degree is at maximum one. In order to preserve the original structure of the paths in the final isomorphic graphs, it is necessary to refine the set of nodes that have to be removed by further coloring. Any node n_j is colored in dark gray if: 1) It is the direct successor of at least two distinct black nodes or, 2) It is the direct successor of the last node n_n^b of an uninterrupted path of black nodes $(n_0^b, \ ... \ , n_n^b)$ and it shares a common direct predecessor n_p with the first node of the path n_0^b. All the blocks having a dark gray successor instead are colored in light gray. The nodes that are still colored in black at the end of the coloring have to be removed. The node removal requires to: 1) Preserve the control flow and nesting levels of the graphs by updating any affected original edge (or back-edge) and, 2) Update the node IDs for preserving the initial incremental labeling scheme, 3) Annotate the



(a) Mapping IR to MIR edges and paths. (b) MIR and binary CFGs isomorphism.

Fig. 3. Steps of the procedure for mapping IR to MIR to binary paths relying first on labels and later on the CFGs isomorphism.

Algorithm 1 Isomorphism Mapping(CFG_{mir}, CFG_{bin})

mapping := ∅
sortByAppearance(N_{mir}); *sortByAddress*(N_{bin})
assignIntegerIds(N_{mir}, N_{bin})
colorNodes(N_{mir}, N_{bin})
CFG^{iso}_{mir},CFG^{iso}_{bin} := *remove&Annotate*(CFG_{mir}, CFG_{bin})
sortAndupdateIntegerIds($N^{iso}_{mir}, N^{iso}_{bin}$)
// Isomorphism completed between CFG^{iso}_{mir} *and* CFG^{iso}_{bin}
for all $(n_i, n_j)^{iso}_{mir} \in E^{iso}_{mir}$ **do**
    $\text{annotatedEdges}_{mir}$:= *getAnnotation*$\big((n_i, n_j)^{iso}_{mir}\big)$
    p_{mir} := *extractPath*$\big((n_i, n_j)_{mir}, \text{annotatedEdges}_{mir}\big)$
    $\text{annotatedEdges}_{bin}$:= *getAnnotation*$\big((n_i, n_j)^{iso}_{bin}\big)$
    p_{bin} := *extractPath*$\big((n_i, n_j)_{bin}, \text{annotatedEdges}_{bin}\big)$
    mapping := mapping $\bigcup$ map(p_{mir}, p_{bin})
return mapping

removed edges information in the updated edges. For example, the graph on the right side of Fig. 3(b) is the result of the removal of the only remaining black node from its colored CFG.

Finally, as shown in Fig. 3(b), a direct match can be identified by relying on the isomorphic versions of the two CFGs. Mixing the information from the node IDs and the annotated edges in the direct match, it is possible to define the mapping between the two original graphs. Two possibilities have to be considered while generating the mapping: 1) An edge is part of the original graph only if it has not been annotated. In this case, the original edge can be mapped with the corresponding edge or path in the other graph. 2) An annotated edge is not part of the original graph because it is the result of some node removal operations. Therefore, if an annotated edge represents a supposed edge (n_i, n_j) in one of the original graphs and its annotation is composed of a list of removed edges $\big((n^r_0, n^r_1),\ \dots\ , (n^r_{n-1}, n^r_n)\big)$, the path $p = \big((n_i, n^r_0),\ \dots\ , (n^r_n, n_j)\big)$ has to be mapped with the corresponding edge or path in the other graph. As a result, the two steps matching approach allows to accurately map the LLVM IR paths to their corresponding binary paths by first translating them into MIR paths.

5 Host-Based Simulation

The detailed workflow designed for executing LLVM IR host-based simulations is described in Fig. 4. The `lli` tool, which is used as the simulator's core, allows executing bitcode on a host machine via Just-In-Time compilation (JIT). The host-execution cannot directly execute bitcode generated for a different target architecture but retargeting the IR module requires only minimal effort. This can be done by implementing an appropriate modification pass for `opt`.

The simulation requires three different inputs: the bitcode to simulate, the CFGs mapping and a source of non-functional information, which is used for updating the simulation results. The simulation bitcode is the result of the linkage of the annotated retargeted IR module with a custom IR library. The annotation

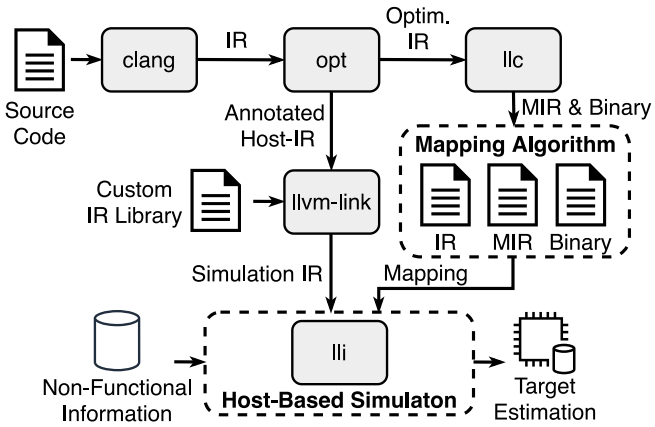


Fig. 4. Combined and detailed workflows proposed for producing the accurate mapping and enabling fast host-based simulations.

consists in the insertion of a function call at the beginning of every basic block. At run-time, the function call forces the simulator to execute the code in the custom library for computing and updating the simulation estimations.

The LLVM IR to binary code mapping is produced as described in Section 4 by analyzing and comparing the different CFGs of the IR, MIR and binary representations. The IR module that is considered during the mapping process is the final IR version optimized by `llc`. This version of the bitcode includes some of the effects of the architecture-dependent optimizations. The MIR module is generated at the end of the last `llc` optimization pass and it includes most of the possible structural changes produced by the back-end.

Finally, the source of non-functional information can be any kind of information that is desired to be considered while simulating. The preference has to be defined in the custom IR library and it can be a model, a database, an external simulator or others.

6 Experimental Evaluation

During compilation, the compiler optimizations are commonly applied in groups of optimization levels rather than individually. Most of them are not independent and applying them in different orders may lead to different results. In our evaluation, we considered all the possible combinations of standard middle-end and back-end optimization levels. Compared with lower optimization levels, the highest level of optimization `-O3` allows observing substantial changes in the structure of the different program representations and is therefore harder to map. For this reason, here we show only the results for the highest level of optimization and omit more accurate results caused by a lower mapping complexity. The evaluation has been conducted on an Intel Core i7-2600K workstation running at 3.4 GHz. An ARM Cortex-A15 with out-of-order superscalar execution and multiple cache levels was chosen as target processor. The analyzed benchmarks are part of the widely-used Mälardalen benchmark suite [7]. The complexity of the

benchmarks, expressed by the quantitative metrics Line Of Codes (LOC) and Cyclomatic Complexity Number (CCN), is reported in Table 1. We excluded the benchmarks that introduce non-available library sources because this may introduce inaccuracies that are not caused by the mapping algorithm's accuracy.

6.1 Mapping Accuracy

A direct validation of the mapping accuracy is not feasible. Therefore, we took an indirect approach that consists of measuring the number of executed instructions and the program execution time and comparing them to the respective results of a simulation based on our mapping approach. The two measured metrics strongly depend on the executed control flow. Consequently, a high level of accuracy in the simulated values indicates a precise mapping of the CFGs.

A fixed number of instructions has been assigned to every binary basic block by statically analyzing the binary file. It is assumed that the execution of a basic block implies the execution of all its instructions starting from its start address. However, the execution time of a basic block can vary depending on the program's execution history due to the possible different timing behavior of the stateful resources included in the processor (e.g. cache memories, pipelines, etc.). Multiple execution times have been consequently measured for every binary basic block, extracted from the target via non-intrusive tracing measurements, allowing to account for the variation caused by the different execution contexts.

The simulator updates both the metrics at run-time relying on the IR execution paths and the mapping information. The evaluation results are summarized in Table 1 showing that the difference between the measured and simulated values is minimal. In most of the cases, the accuracy is close to 100%. This implies a high level of accuracy of the simulation results and consequently of our mapping approach. One reason for the deviation in the results is that the ARM instruction set includes conditional execution instructions that partially invalidates our assumption on the complete execution of a basic block. Nevertheless, the effectiveness of the mapping is proven by the accuracy of the simulation results.

Compared to other mapping approaches the one we propose is fully automatic and achieves a high level of precision. However, since it relies on the information contained in the LLVM MIR, it requires the analyzed programs to be compiled with LLVM and cannot simply be ported to other compilers. Furthermore, for us it has proven to be sufficient to match sequence of edges of the IR CFG to the binary CFG's paths. For other approaches requiring the mapping to be performed on basic blocks or instructions, this may not be suitable.

6.2 Host-Based Simulation Speed

A second evaluation objective has been to determine the potential simulation speed. We have measured the time required for simulating the benchmarks. The simulation speed is measured in Million of simulated Instructions Per Second (MIPS). In Fig. 5 the results of our measurements are shown. For consistency reasons, the chart shows only the results for benchmarks compiled with `-O3`

Table 1. Measured mapping's accuracy with `-O3` optimization level.

Benchmark	Metrics		Instructions Count			Execution Time (us)		
	LOC	CCN	Measured	Simulated	Accuracy	Measured	Simulated	Accuracy
bs	144	9	51	51	**100%**	7	7	**100%**
bsort100	128	8	45949	44286	**96.38%**	2380	2301	**96.68%**
cnt	267	3	1525	1525	**100%**	165	165	**100%**
crc	128	9	11661	11661	**100%**	635	635	**100%**
duff	86	10	537	537	**100%**	37	37	**100%**
edn	285	4	84992	84301	**99.19%**	5989	5945	**99.27%**
fdct	239	3	1834	1834	**100%**	159	159	**100%**
fir	276	5	133588	133588	**100%**	9270	9340	**99.24%**
insertsort	92	8	1108	1083	**97.74%**	92	90	**97.83%**
ludcmp	147	14	1512	1506	**99.61%**	235	231	**98.30%**
matmult	163	4	35229	35229	**100%**	3015	3040	**99.17%**
ndes	231	11	28625	28566	**99.79%**	1502	1489	**99.13%**
prime	47	4	4233	4223	**99.76%**	449	448	**99.78%**
qsort_exam	121	15	851	867	**98.12%**	79	76	**96.23%**
qurt	166	5	514	514	**100%**	135	135	**100%**
select	144	16	368	368	**100%**	34	34	**100%**
sqrt	77	5	447	447	**100%**	124	124	**100%**
st	177	4	70067	70067	**100%**	4548	4518	**99.34%**
ud	163	11	1205	1161	**96.35%**	109	106	**97.24%**

optimization level. We considered only the results for the benchmarks that ensure simulating more than ten thousand ARM instructions. The maximum observed simulation speed value has been of 69 MIPS while simulating the benchmark `crc`. The same benchmark compiled with `-O2` showed the highest value of our entire evaluation reaching the value of 90 MIPS. We observed that the simulation speed is directly related to the number of simulated instructions. The simulation of a substantial number of instructions reduces the minimal overhead due to the basic blocks instrumentation and the JIT compilation. We are confident that this will be advantageous for real applications considering that they typically execute more instructions than the used benchmarks. We also observed that the type of instructions in a program influences the simulation speed. Arithmetic intensive benchmarks showed higher speed than others.

An additional evaluation has been conducted for comparing the performance [4] of our host-based simulation approach with the well-known and public available gem5 simulator [3]. We compared the amount of time required for simulating the benchmarks with our simulation approach against the time required by gem5. We compiled gem5 configuring the fast mode of the full system simulation mode [10]. The histogram's columns in Fig. 5 show the speedup resulting from simulating the benchmarks with our approach and requiring a logarithmic scale. The light gray columns represent the comparison results when gem5 only simulates ARM instructions without timing or physical resource consideration. The darker columns show the comparison with a full timing gem5 simulation. We observed a maximum speedup of 20 comparing the simulation of the `edn` benchmark without timing considerations. The maximum observed speedup increased to 527 simulating the `crc` benchmark with gem5 and by enabling the consideration of the timing behavior of caches, pipeline and branch predictor. The average

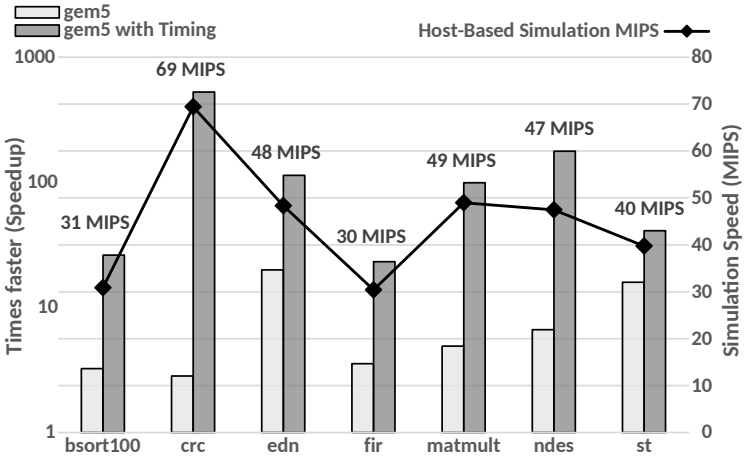


Fig. 5. Simulation speedup compared with the performance of the gem5 simulator executed with and without timing considerations.

observed speedup in case of timing simulation is 144. The results demonstrate that our approach allows executing accurate simulations requiring an amount of simulation time that is orders of magnitude shorter than simulating with gem5.

7 Summary and Conclusions

In this paper, we presented a new approach for automatic mapping of LLVM IR to binary embedded code for the purpose of host-based simulations in support of system design decisions. The mapping approach relies on the information extracted from the intermediate LLVM MIR representation that is internal to the compiler without requiring any modification. Overall experiments results show a high level of accuracy even in presence of aggressive compiler optimizations (average error smaller than 2%). The conducted evaluation activities prove that our mapping can be utilized for evaluating the execution cost of embedded programs in addition to the quantification of code coverage metrics [8] at different program representations. Furthermore, the proposed LLVM IR host-based simulation approach shows a substantial speedup of several orders of magnitudes compared with the gem5 simulator (144 average observed speedup).

Future work will focus on the further evaluation of different target architectures and benchmarks. Prospectively, we intend to refine the granularity of the mapping algorithm for identifying a direct mapping between the LLVM IR and binary instructions. Furthermore, a straightforward extension has been planned for additionally mapping the source code to LLVM IR. The extension will enable the possibility of source-level simulations and consequently improving the simulation speed while keeping the showed level of accuracy.

Acknowledgements This work has been partially supported by the German Federal Ministry of Education and Research (BMBF) within the projects COMAPCT under grant 01|S17028C and progressivKI under grant 19A21006M.

References

1. Aho, A., Lam, M., Sethi, R., Ullman, J., Cooper, K., Torczon, L., Muchnick, S.: Compilers: Principles, Techniques and Tools. Addison-Wesley 2nd edition (2007)
2. Becker, M., Pazaj, M., Chakraborty, S.: WCET analysis meets virtual prototyping: Improving source-level timing annotations. In: 22nd International Workshop on Software and Compilers for Embedded Systems (2019)
3. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., et al.: The gem5 simulator. ACM SIGARCH computer architecture news (2011)
4. Butko, A., Garibotti, R., Ost, L., Sassatelli, G.: Accuracy evaluation of gem5 simulator system. In: International workshop on reconfigurable and communication-centric systems-on-chip (ReCoSoC). IEEE (2012)
5. Chakravarty, S., Zhao, Z., Gerstlauer, A.: Automated, retargetable back-annotation for host compiled performance and power modeling. In: International Conference on HW/SW Codesign and System Synthesis. IEEE (2013)
6. Cornaglia, A., Hasan, M.S., Viehl, A., Bringmann, O., Rosenstiel, W.: JIT-based context-sensitive timing simulation for efficient platform exploration. In: 25th Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE (2020)
7. Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B.: The Mälardalen WCET benchmarks: Past, present and future. In: 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010) (2010)
8. Jahić, J., Kuhn, T., Jung, M., Wehn, N.: BOSMI: a framework for non-intrusive monitoring and testing of embedded multithreaded software on the logical level. In: 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (2018)
9. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization. IEEE (2004)
10. Lowe-Power, J., Ahmad, A.M., Akram, A., Alian, M., et al.: The gem5 simulator: Version 20.0+. arXiv preprint arXiv:2007.03152 (2020)
11. Lu, K., Müller-Gritschneder, D., Schlichtmann, U.: Hierarchical control flow matching for source-level simulation of embedded software. In: 2012 International Symposium on System on Chip (SoC). IEEE (2012)
12. Matoussi, O., Pétrot, F.: A mapping approach between IR and binary CFGs dealing with aggressive compiler optimizations for performance estimation. In: 23rd Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE (2018)
13. Stattelmann, S., Bringmann, O., Rosenstiel, W.: Dominator homomorphism based code matching for source-level simulation of embedded software. In: International Conference on HW/SW Codesign and System Synthesis (2011)
14. The LLVM Compiler Infrastructure: Machine IR (MIR) Format Reference Manual, https://llvm.org/docs/MIRLangRef.html
15. The LLVM Compiler Infrastructure: The LLVM Target-Independent Code Generator, https://llvm.org/docs/CodeGenerator.html
16. Wang, Z., Henkel, J.: Accurate source-level simulation of embedded software with respect to compiler optimizations. In: Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE (2012)
17. Wang, Z., Herkersdorf, A.: An efficient approach for system-level timing simulation of compiler-optimized embedded software. In: 46th Design Automation Conference (DAC). IEEE (2009)