Advancing Source-Level Timing Simulation using Loop Acceleration

Joscha Benz, Christoph Gerum and Oliver Bringmann
Embedded Systems Department, University of Tübingen
{benz,gerum,bringmann}@informatik.uni-tuebingen.de

***Abstract*—Source-level timing simulation (STLS) is an important technique for early examination of timing behavior, as it is very fast and accurate. A factor occasionally more important than precision is simulation speed, especially in design space exploration or very early phases of development. Additionally, practices like rapid prototyping also benefit from high-performance timing simulation. Therefore, we propose to further reduce simulation run-time by utilizing a method called loop acceleration. Accelerating a loop in the context of SLTS means deriving the timing of a loop prior to simulation to increase simulation speed of that loop. We integrated this technique in our SLTS framework and conducted an comprehensive evaluation using the Mälardalen benchmark suite. We were able to reduce simulation time by up to 43% of the original time, while the introduced accuracy loss did not exceed 8 percentage points.**

***Index Terms*—timing simulation, SLTS, loop acceleration**

I. Introduction

There is a constantly increasing need for speed and flexibility during development of embedded software. In addition, systems and software become more complex. Therefore, developers need powerful tools to be able to create products that comply with functional and non-functional requirements such as timing constraints. Source-level timing simulation (SLTS), also called host-compiled simulation, serves these needs, as it allows very fast and accurate simulation of timing behavior. The high speed of SLTS is achieved by compiling and running the software to analyze on a different and much faster machine than the target. For that purpose, it is necessary to be able to reconstruct execution paths through the target binary during simulation. The prerequisites of SLTS are therefore twofold: 1) create a matching between binary- and source-level control-flow and 2) instrument the original source-code according to that matching with timing information. This instrumentation is called *path simulation code*, as it is used to simulate control-flow of the target binary during host-compiled simulation. Early approaches to SLTS have instrumented every basic block at the cost of a large overhead. Thus, recent source-level simulations (SLS) use optimizations to reduce the number of instrumented BBs and, consequently, the run-time of the instrumented program without significant impact to preciseness [1]. Figure 1b shows example source code with a minimum number of instrumented basic blocks. Regarding loops, another opportunity to increase simulation performance arises: removing instrumentation points that are part of a loop. For example, Figure 1b shows source code of a loop, which contains multiple instrumentation points. Considering the iteration count (*loop bound*) of this loop, `simulate_bb_2` is called 400 times during simulation. Thus, assuming statically available bounds, one can expect to further reduce simulation run-time by removing `simulate_bb_2` and `simulate_bb_3`. Moreover, we expect a compiler to be able to optimize loops more aggressively once instrumentation points are removed. The latter can additionally improve simulation speed. In the scope of this paper we denote statically deriving the execution time of a loop and annotating the results back to source-code prior to simulation as *loop acceleration*.

Based on this idea, we make the following contributions as part of this work

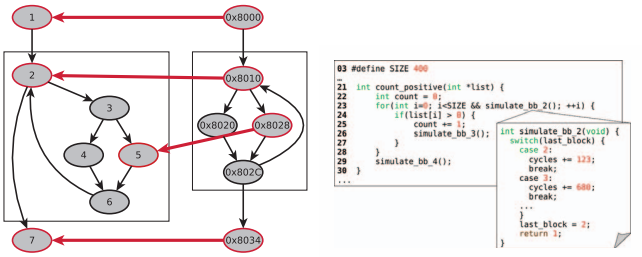
- An algorithm to accelerate loops and remove obsolete instrumentation points
- A heuristic for accelerating loops while containing the introduced loss of accuracy
- An experimental evaluation of our approach based on a widely used benchmark suite

The rest of this paper is structured as follows. Related work is presented in Section II. Fundamental concepts are covered in Section III. Section IV discusses the contributions of this work in detail. Experimental evaluation and the corresponding results are accounted for in Section V. Section VI concludes this work and briefly discusses future research.

II. Related Work

Much work on source-level timing simulation exists with emphasis on different problems. Several approaches to control-flow mapping have been proposed, for example [6], [8]. Both techniques focus on creating a matching that is as accurate as possible, rather than reducing instrumentation. Other research has been conducted with attention to increasing accuracy, speed or both [4], [5]. The work presented in [4] focuses on integrating cache simulation into SLTS, while keeping performance as high as possible and achieving high accuracy. Simulation of power and timing with instrumentation on IR- rather than source-level was proposed in [5]. A more recent work utilizes machine learning for power and performance estimation [9] achieving quite high simulation performance and accuracy. On the other hand, this approach requires availability to a, not necessarily physical, model of the hardware

This contribution is funded as part of the CONFIRM project (project label 16ES0564-70) within the research program ICT 2020 by the German Federal Ministry of Education and Research (BMBF) and is supported by the industrial partners Infineon Technologies AG, Robert Bosch GmbH, Intel Deutschland AG, and Mentor Graphics GmbH.



(a) An example matching (b) Example path simulation code

Fig. 1. Two steps of SLTS: matching and path simulation code generation

with access to performance counters to obtain training data. In contrast, SLTS as introduced in the previous section does not suffer this restriction. The publications mentioned so far did not focus on lowering the amount of instrumentations to increase simulation speed. In [1] an algorithm to reduce the number of instrumented source-level basic blocks is proposed. More specifically, the suggested technique allows to find a minimum set of instrumentation points necessary to recreate binary control-flow during simulation. The work presented in this paper aims at improving the simulation speed of state of the art approaches, which already use instrumentation point reduction, such as [1].

III. Fundamentals

A. Source-Level Timing Simulation

Source-level timing simulation requires several analysis steps to be run prior to the actual execution. Three of these are of interest in the scope of this work. First, matching of binary- and source-level control-flow, as shown in Figure 1a. Second, path analysis enumerates all paths between pairs of matched binary basic blocks. Note that a path may have multiple possible successors. To identify which of these is executed during simulation, it is sufficient to record the execution of the last basic block in a path [1], which is the main purpose of path simulation code. Figure 1b shows an example of the latter. When `simulate_bb_2` is called during simulation, the number of overall cycles is increased depending on the last instrumented basic block encountered. In this example, BB 2 may be preceded by itself or by BB 3. It also may be preceded by any number of basic blocks outside of `count_positive`. As shown in Figure 1b, the generated code contains information about the execution time of each path, that is accumulated throughout simulation. Next, the original source code is instrumented according to the matching created in the first phase. Thus, each matched source-level basic block is an instrumentation point.

B. Loop Acceleration

Abstract loop acceleration is used in WCET analysis to statically derive an abstract state that represents values of variables after the execution of a loop. In contrast, loop acceleration is applied during path analysis in this work. More specifically, a loop L is accelerated by first enumerating all possible paths from loop-header to a loop-latch. Such a path consists of all basic blocks in-between, including those part of other loops or functions as well. We denote these paths as *full loop-paths* for the reminder of this paper. Next, one of these paths is chosen, for example the longest w.r.t the number of basic blocks. Then, this path is appended to itself $L.upper\_bound - 1$ times. Finally, the timing of L is derived by calculating the execution time of the unrolled path. As a result, all instrumentation points part of an accelerated loop can be removed.

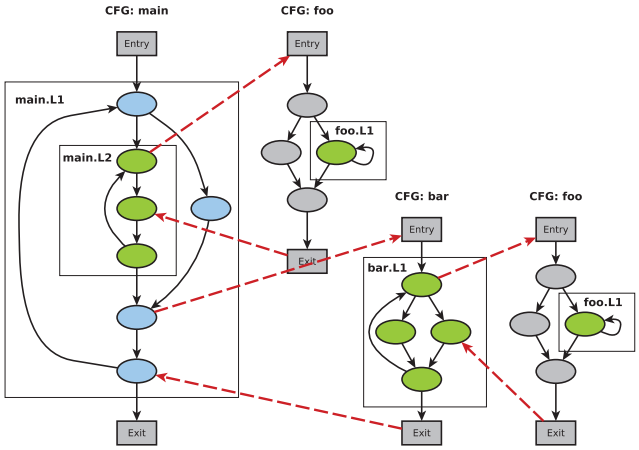


Fig. 2. Exemplary Call and Loop Dependencies

On the other hand, accelerating loops may decrease the accuracy of SLTS, as loop bounds usually are conservative. Moreover, there may be multiple paths connecting the same pair of matched basic blocks as a result of loop unrolling. For example, in Figure 2 a full loop-path of `main.L1` may either contain `main.L2` or not, while one expects the execution time of the latter to be much lower. Therefore, additional loss of accuracy is introduced by choosing one of these paths to be part of path simulation code. In consequence, we heuristically choose loops for acceleration, such that the decrease of accuracy can be controlled.

C. Execution Contexts

In general the execution time of an instruction depends on the state of the microarchitecture, for example caches or the pipeline. Therefore, it depends on the execution history leading to an instruction. In timing simulation context-sensitivity of execution times is exploited to greatly improve the precision of timing predictions [2], [3].

IV. Methodology

In the following, we first give a short overview of our method and then describe each step in detail. Loop acceleration, as realized in this work, consists of three phases: 1) choosing loops suitable for acceleration, 2) actual loop acceleration and 3) adjusting instrumentation points. The last step is necessary to make sure that control-flow reconstruction is complete and correct after loops have been accelerated.

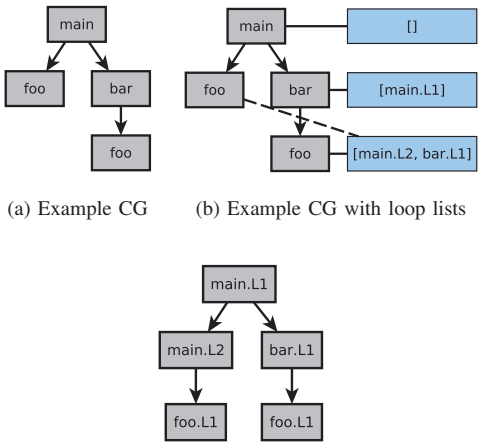
Following this, we present a heuristic to prevent large accuracy loss due to acceleration.

A. Loop Selection

A loop has to satisfy several requirements to be suitable for acceleration. First, statically derivable upper bounds of a loop and its sub-loops are mandatory. Additionally, unrolling may also depend on a loop that is not a sub-loop. For example, Figure 2 shows such a dependency, as `main.L2` depends on `foo.L1` due to a call made from the body of `main.L2`. We denote this kind of dependency as *loop dependency* for the rest of this paper. Note that we duplicated the CFG of `foo` for the sake of clarity. In general, it is possible for `foo.L1` to depend on additional loops. For instance, `bar.L1`, being a dependency of `main.L1`, has one of its own. Thereby, loop dependency is a transitive relation. Thus, loop selection consists of: 1) analyzing loop dependencies and 2) filtering loops by availability of upper bounds.

1) Loop-Dependency Analysis: As loop dependencies can result from calls to other functions, we use a call-graph (CG) for analysis. More specifically, the CG is iterated depth-first, while each node is associated with a list of loops. If a loop L is mapped to a node N, there exists a call from the body of L to the function represented by N. On visiting a CG node, each call-site (CS) is checked to be part of a loop. If that is the case, the inner-most loop containing a CS is associated with the CG node of the called function. For example, Figure 3b shows the list of loops associated with each node of the CG in Figure 3a. As `bar` is only called from the body of `main.L1`, the corresponding list solely consists of the latter. Moreover, `foo` is called from `main.L2` as well as `bar.L1`. Thus, the associated list contains both loops. Note that we expect loop dependency to be anti-symmetric. Thus, we currently do not support accelerating loops with cyclic dependencies. When visiting a CG node N that is associated to a loop L, all loops part of the function corresponding to N are marked as a dependency of L. In addition, each caller of N is marked as depending on these loops. Finally, this algorithm yields two *dependency maps*. One relates a loop to a list of all its direct loop dependencies. The other associates a function F to all loops contained by callees of F.

Next, we create a list lst_accel of all loops ordered such that all dependencies of a loop L appear before L. To that end, we first collect all loops that are not depended-on by other loops. We denote these as *free loops*. Given that loop dependency is a transitive relation, we may regard a dependency map dep as a tree. More specifically, each free loop can be considered as a root of such a tree. Additionally, the children of a node N in a dependency-tree are defined by the list $dep(N)$. See Figure 3c for the loop-dependency tree of `main.L1`. As the first step in creating lst_accel, we iterate the dependency-tree of each free loop in post-order. To be more precise, for each dependency-tree, we create a list of loops in the same order they are visited during iteration. The list resulting from iterating the tree in Figure 3c is: $[\texttt{foo.L1}, \texttt{main.L2}, \texttt{bar.L1}]$. Note that `foo.L1` is added once, since adding it twice would be redundant. Finally, each list contains loops ordered such that a loop L appears after all its loop dependencies. We create lst_accel by applying k-ways merge, while each loop is added at most once.



(a) Example CG (b) Example CG with loop lists

(c) Loop-dependency tree of main.L1

Fig. 3. Example Call Graph, annotated CG and Loop-dependency Tree

Lastly, the derived list is filtered by removing all loops that can not be accelerated. There are several reasons for a loop to be removed: no upper bound available or removal of a dependency. Additionally, assume multiple loops containing a call to the same function and one of these is filtered. In that case, all other loops have to be removed as well. Otherwise, control-flow reconstruction fails during simulation of one of the accelerated loops. In summary, the result of loop selection and loop-dependency analysis is a list of all loops ordered for acceleration.

B. Loop Acceleration

As mentioned earlier, accelerating a loop includes explicitly enumerating all full loop-paths and unrolling these afterwards. Explicitly enumerating all possible paths in a directed graph is undecidable in general, as paths may be infinitely long due cycles. Since we can detect loops and know the bounds of those we are accelerating, the problem to be solved can be reduced to enumerating all paths in a directed acyclic graph (DAG). This problem still has exponential complexity. In addition, overhead can be reduced by enumerating the CFG of a function that is called by multiple accelerated loops only once. To that end, call dependencies between function need to be known, as callees need to be enumerated before their callers. Figure 2 shows exemplary call dependencies where the CFG of `foo` needs to be enumerated before `bar` and consequently before `main`. In summary, three steps are necessary to realize loop acceleration: 1) call dependency analysis, 2) explicit path enumeration and 3) loop-path unrolling.

1) Call-Dependency Analysis: Call-dependency analysis is straight-forward based on the information provided by a call-

graph, whereas transitivity of call dependency has to be considered as well. The CG is traversed depth-first to derive a mapping of direct dependencies, which associates a caller with its callees. Next, similar to loop-dependency analysis, a list of CFGs is generated based on direct dependencies. Consequently, this list is ordered such that a caller appears after all its callees. Hence, it can be used to hierarchically enumerate all paths needed during loop acceleration.

2) Hierarchical Path Enumeration: As mentioned in Section III, context-sensitivity of execution times can be used to greatly improve the accuracy of timing simulation. Thus, our approach supports both simulation modes: with and without context-sensitivity. All of the algorithms presented in the following work on a context-sensitive ICFG. Such a graph allows to track the current context during iteration and to context-sensitively retrieve information such as successors of a basic block or bounds of a loop. However, we currently do not have sophisticated context-sensitive analyses. Thus, we focus on context-insensitive simulation in the scope of this work.

Figure 2 shows three functions, namely: `main`, `foo` and `bar`. It further shows four loops, as well as all loop and call dependencies in this example. Loop dependencies are not visualized plainly but implicitly. In this example, hierarchical path enumeration is first applied to `foo.L1` and `foo`, followed by enumerating all paths of `main.L2`. Next, all paths of `bar.L1` and `bar` are enumerated. Finally, all full loop-paths of `main.L1` can be created. This way, it is possible to reuse paths during enumeration of CFGs at a lower level.

Based on these observations, we implement hierarchical path enumeration as follows. We start by iterating over the list of loops yielded by loop selection. For each loop L, we use the results from call-dependency analysis to get a correctly ordered list of CFGs that have to be enumerated before L. Subsequent to handling this list, all paths of L are enumerated. Other loop dependencies do not need to be explicitly considered during this process, as the list of loops returned by loop selection is ordered accordingly.

Going back to the example in Figure 2 our algorithm proceeds as follows, assuming statically available bounds. As `main.L1` is the only free-loop in this example, the list returned by loop selection immediately results from the dependency-tree in Figure 3c. Thus, `foo.L1` is handled first. Processing a loop consists of two steps: first each CFG called from that loop is enumerated if necessary. Next, the loop itself is taken care of. Enumerating all paths of a loop L starts with its header, denoted as `L.head`, while intermediate paths are memorized using a stack. To create new paths, the one currently on top is popped off. For each successor of that paths' last basic block a new path can be created and pushed on the stack. However, not all successors can be simply appended for path creation. More specifically, there are three cases in which new paths are created differently. First, a successor may be part of an already enumerated sub-loop L'. In that event, new paths are created by appending all unrolled paths from L'. Alternatively, the next basic block may be part of a different function due to a call. Thus, path creation is done analogously to the previous case. Finally, a successor may already be part of the current path. In that case, no path is created, as multiple occurrences of a basic block indicate a loop that was not selected for acceleration. Enumerating all paths through a CFG is executed analogously. However, a full CFG-path starts with an *entry*-block and ends with an *exit*-block, shown as rectangles in Figure 2.

3) Loop-Path Unrolling: Following enumeration, loop-paths are unrolled heuristically to account for the entire execution of a loop. To this end, each enumerated path is concatenated to itself as often as the corresponding loop bound requires. Therefore, it is mandatory for all paths to end with a loop-latch. These unrolled paths do not represent all possible paths through a loop, as different iterations may have alternating paths from header to a latch. However, creating all unrolled paths w.r.t control-flow across different iterations is computationally very expensive. Thus, just concatenating full loop-paths to themselves is a heuristic to contain that complexity. Subsequent to unrolling, a path is extended to end with a loop-exit if necessary. This is required for instrumentation-point adjustment, which is discussed next.

C. Instrumentation-Point Adjustment

Following enumeration and unrolling, the derived paths need to be integrated into the results of path analysis to be utilizable during path simulation code generation. To that end, a path must start and end with a matched basic block. Hence, we realize integration using already existing simulation paths as created during path analysis. More precisely, we are interested in original simulation paths that enter or leave an accelerated loop. For example, in Figure 1a path $(\texttt{0x8028}, \texttt{0x802c}, \texttt{0x8034})$ leaves the corresponding loop in Figure 1a, while path $(\texttt{0x8000}, \texttt{0x8010})$ enters it. Thus, we first collect all paths leaving or entering an accelerated loop. Next, each path is split at a boundary of the corresponding loop, being either the loop-header or a loop-exit. This process yields all paths entering/leaving a loop, ending with a loop header or starting with a loop-exit respectively. In the following we denote these as loop-entry or loop-exit paths. These paths can now simply be concatenated/appended to the existing acceleration paths to create valid simulation paths. Note that this procedure is guaranteed to result in valid and complete path simulation code, assuming the latter possessed these properties prior to acceleration. Validity in terms of correct control-flow is assured, since concatenating paths is conducted w.r.t the successors of a basic block. As mentioned earlier, loop-paths are not complete, but only w.r.t control-flow inside a loop. Two properties of the accelerated paths allow derivation of completeness. First, all loop paths start with the loop-header and any loop-entry path must enter a loop via the loop-header. Hence, any such path can be concatenated with any accelerated path. Secondly, there is a path from loop-header to each loop exit and as any loop-exit path must contain one of these exits blocks, it is possible to find an accelerated path for each possible loop-exit path. Therefore, all original

simulation path entering and leaving an accelerated loop are part of path simulation code after acceleration as well.

D. Acceleration Heuristic

Our acceleration heuristic consists of two simple formulas. First, the difference between upper and lower bound of a loop L is considered during loop selection:

$$\frac{L.upper - L.lower}{L.upper} \cdot 100 \leq max_{loss},$$

while max_{loss} is a user-provided ratio. Second, the latter is used to restrict the allowed distance between the longest and shortest unrolled loop-path:

$$\frac{max(len(path)) - min(len(path))}{max(len(path))} \cdot 100 \leq max_{loss}.$$

Note that $len(path)$ denotes the number of BBs in a path. In addition, we track the sum of these distances, which must not exceed the user-provided percentage as well. Thus, any loop causing the accumulated and estimated inaccuracy to exceed the threshold is not accelerated. In the following, we denote this user-supplied number as *expected inaccuracy*.

V. Experimental Evaluation

In this section we present and discuss the evaluation of our proposed approach. We first examine setup and parameters of the experimental evaluation. Afterwards, we review our findings and their implications regarding our work.

To assess the loss of accuracy introduced by loop acceleration, we first ran a selection of benchmarks from the Mälardalen benchmark suite [7] on a Cortex-M0+ board from Freescale. More specifically, we used a FRDM-KL25Z development board, with a main clock speed of 20971520 Hz. Using this frequency, the microcontroller does not have to wait during communication with the on-board flash. This is necessary, as our static timing model for the Cortex-M0+ is rather simple. We made some changes to the selected benchmarks, which we lay out after providing rationale for that decision. One of two issues regarding the original sources is missing HW support for floating-point (FP) arithmetic and integer division. The handwritten assembly emulating this functionality results in highly optimized and irregular binary-level control-flow. Therefore, we replaced any integer division with manually written C-code, while we discarded any benchmark with FP arithmetic. In consequence `cnt`, `compress`, `jfdctint` and `matmult` were changed regarding this issue. Next, some of the benchmarks consist of initialization routines, the effects of which are persistent across consecutive executions. Therefore, the corresponding execution times differ between initialized and non-initialized runs. As our framework always simulates a benchmark entirely, we changed `crc` such that initialization is non-persistent. In addition, all benchmarks were annotated with best- and worst-case loop bounds w.r.t the given input at source-level. These bounds were extracted by our framework to be used for loop acceleration. To be able to apply loop acceleration to a larger benchmark we combined `adpcm_enc` and `adpcm_dec` from the TACLE benchmark suite [10]. The resulting application consist of about 2000 lines of code.

All benchmarks were compiled using *GCC ARM Embedded 6 update 2* for the target and *GCC 6.3* for the host-machine. The latter consists of an Intel Core i5-6500 CPU and 16 GB RAM. As host operating system Scientific Linux 7 was used, while the target ran benchmarks directly (bare-metal). Furthermore, interrupts were disabled during execution.

We conducted two different experiments on the host. First, we simulated each benchmark to get execution time predictions. Next, we ran benchmarks to measure run-time of simulation with and without loop acceleration. We used the tool `time` to measure the entire execution time of a simulation run, including the time a process was preempted. The following simulation modes were used for both experiments: normal, loop acceleration (accel) and loop acceleration with heuristic (accel+heu). Note that the corresponding abbreviations are used in all figures in this section. Further note that we used an allowed expected inaccuracy of 10% for heuristic acceleration.

To get execution time predictions we executed our simulation once for each benchmark and simulation mode. In contrast, to collect simulation run-times, we ran 10 simulations for all modes and benchmarks. We derived a single value for each benchmark by calculating the median of these run-times. As our host-machine is very powerful compared to an embedded system, an individual execution of any benchmark only took a fraction of a second. Moreover, noise is caused by the operating system and multi-tasking. Thus, a single simulation run consisted of multiple consecutive executions of a benchmark to account for said noise. As a result, the execution times of benchmarks on our host-machine ranged from 1-23 seconds.

To measure the execution time of a benchmark on real hardware, we used the MCUs general purpose IO and a logic analyzer. More specifically, a certain pin was set to *HIGH* immediately before and to *LOW* directly after execution of a benchmark. As bare-metal execution is subject to noise as well, we ran each benchmark at least 750 times. The reference time used for all comparisons is the average of these execution times.

A. Results

Figure 4 shows the relative error of different simulation modes w.r.t the execution times measured on hardware. We calculated this inaccuracy as follows:

$$error = \frac{t_{pred} - t_{exec}}{t_{exec}} \cdot 100,$$

while t_{pred} denotes the execution time predicted by our simulation and t_{exec} the measured execution time on hardware. On the other hand, Figure 5 shows achieved reduction of simulation time by using loop acceleration compared to normal simulation. We calculated this performance gain as follows:

$$reduction = \frac{t_{sim} - t_{acc}}{t_{sim}} \cdot 100.$$

Note that t_{acc} may denote any of the used simulation modes.

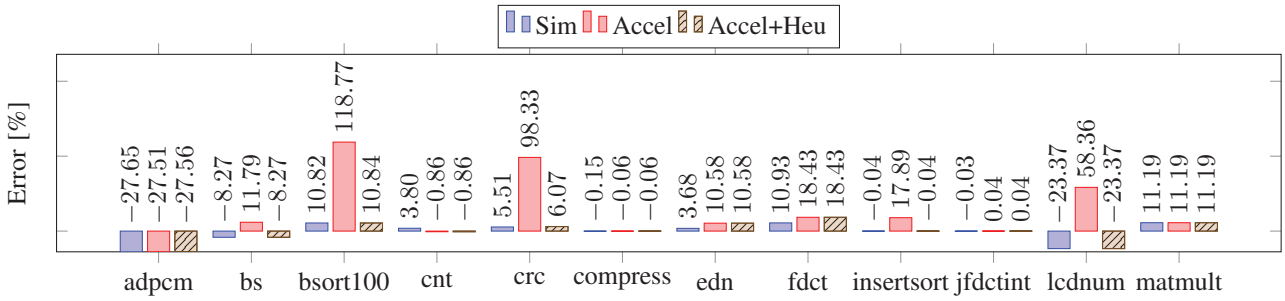


Fig. 4. Loss of Accuracy w.r.t Execution on Hardware

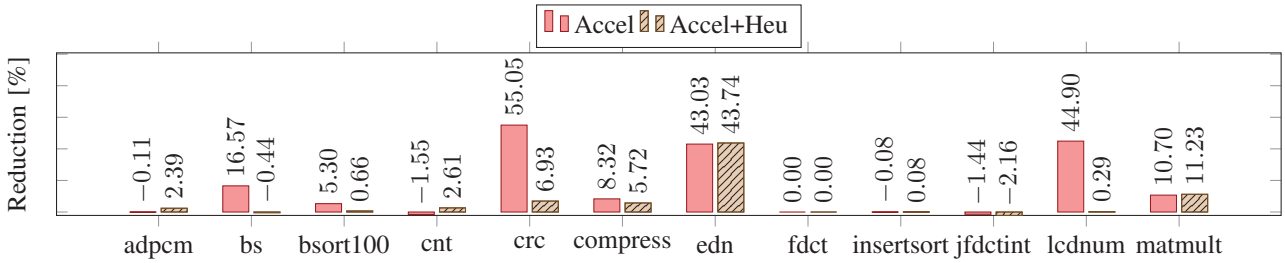


Fig. 5. Simulation-Time Reduction w.r.t Simulation without Loop Acceleration

First of all, our results indicate 2 things: 1) loop acceleration is generally capable of achieving quite large simulation time reduction, while 2) increasing the inaccuracy of the prediction considerably in some cases. The letter is to be expected, which is why we implemented a simple heuristic to choose loops for acceleration. Figure 4 shows that this heuristic effectively prevents large simulation errors, as for example in case of `bsort100`, `crc`, `insertsort` or `lcdnum`. However, the gain in simulation speed is reduced as well. Nonetheless, loop acceleration is capable of decreasing the simulation time between 5% and 43% for some benchmarks, while keeping the introduced inaccuracy in acceptable bounds.

For example, `crc`, `compress`, `edn` and `matmult` experienced accuracy loss in the range of about 1 to 8 percentage points, while achieving decreasing simulation time between 5% and 43%. Especially `mamult`, which did not encounter any accuracy loss but a simulation time reduction of 11%.

In addition, it is to be noted that acceleration may lead to negligible slowdown of overall simulation in some cases. In these cases, the accelerated loops accommodated just for a very small part of the entire execution of a benchmark. Additionally, removing `simulate_bb` procedures from loops can cause an increase of L1 instruction cache misses. Furthermore, we expect these effects to cause the behavior shown by `jfdctint`. Moreover, variations in execution time of a process running on linux in the range of less than 3% can be attributed to different states of the microarchitecture or noise produced by the operating system.

VI. Conclusion

We presented a way to use loop acceleration to further accelerate SLTS along with a heuristic capable of containing the corresponding loss of accuracy. In addition, our evaluation indicates that next steps to further accelerate simulation need to incorporate the effects of instrumentation on microarchitectural state. More specifically, choosing instrumentation points and ordering bodies of corresponding functions w.r.t instruction cache behavior is left as future work.

References

[1] S. Schulz, O. Bringmann, Accelerating Source-Level Timing Simulation, Proceedings of the 2016 Conference on Design, Automation & Test in Europe, pp.1574–1579, 2016.

[2] S. Ottlik, C. Gerum, A. Viehl, O. Bringmann, Context-Sensitive Timing Automata for Fast Source Level Simulation, Proceedings of the 2017 Conference on Design, Automation & Test in Europe, pp.512–517, 2017.

[3] S. Ottlik, S. Stattelmann, A. Viehl, W.Rosenstiel, O. Bringmann, Context-Sensitive Timing Simulation of Binary Embedded Software, Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp.512–517, 2014.

[4] Z. Wang, J. Henkel, Fast and Accurate Cache Modeling in Source-Level Simulation of Embedded Software, Proceedings of the 2013 Conference on Design, Automation and Test in Europe, pp.587–592, 2013.

[5] S. Chakravar, Z. Zhao, A. Gerstlauer, Automated, Retargetable Back-Annotation for Host Compiled Performance and Power Mode, Proceedings of the Nineth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2013.

[6] Z. Wang, J. Henkel, Accurate Source-Level Simulation of Embedded Software with Respect to Compiler Optimizations, Proceedings of the 2012 Conference on Design, Automation and Test in Europe, pp.382–387, 2012.

[7] J. Gustafsson, A. Betts, A. Ermedahl, B. Lisper, The Mälardalen WCET Benchmarks: Past, Present And Future, 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010), pp.136–146, 2010.

[8] S. Stattelmann, O. Bringmann, W. Rosenstiel, Fast and Accurate Source-Level Simulation of Software Timing Considering Complex Code Optimizations, Proceedings of the 48th Design Automation Conference, pp.486–491, 2010.

[9] X. Zheng, L. K. John, A. Gerstlauer, Accurate Phase-Level Cross-Platform Power and Performance Estimation, Proceedings of the 53rd Annual Design Automation Conference , Article No. 4, 2016.

[10] H. Falk et al., TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research, Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016), pp.1–10, 2016.