Context-Sensitive Timing Automata for Fast Source Level Simulation

Sebastian Ottlik*, Christoph Gerum†, Alexander Viehl*, Wolfgang Rosenstiel*†, Oliver Bringmann*†

*FZI Research Center for Information Technology
Haid-und-Neu-Str. 10–14
D-76131 Karlsruhe, Germany

†University of Tübingen
Sand 13
D-72076 Tuebingen, Germany

Abstract—**We present a novel technique for efficient source level timing simulation of embedded software execution on a target platform. In contrast to existing approaches, the proposed technique can accurately approximate time without requiring a dynamic cache model. Thereby the dramatic reduction in simulation performance inherent to dynamic cache modeling is avoided. Consequently, our approach enables an exploitation of the performance potential of source level simulation for complex microarchitectures that include caches. Our approach is based on recent advances in context-sensitive binary level timing simulation. However, a direct application of the binary level approach to source level simulation reduces simulation performance similarly to dynamic cache modeling. To overcome this performance limitation, we contribute a novel pushdown automaton based simulation technique. The proposed context-sensitive timing automata enable an efficient evaluation of complex simulation logic with little overhead. Experimental results show that the proposed technique provides a speed up of an order of magnitude compared to existing context selection techniques and simple source level cache models. Simulation performance is similar to a state of the art accelerated cache simulation. The accelerated simulation is only applicable in specific circumstances, whereas the proposed approach does not suffer this limitation.**

I. Introduction

Simulations are an important tool for embedded systems research and development. Novel techniques that address the need for fast, yet accurate simulations have been proposed by recent research. For simulating software executions on a target system, so called source level simulations (SLS) have shown a high potential in terms of simulation performance while offering a reasonable level of accuracy. In SLS the software source code is augmented to enable a simulation of non-functional properties — in particular timing. As the augmented code can be compiled directly for a simulation host, the simulation can be highly efficient if the overhead for the additional code is sufficiently low. However, current approaches to SLS require dynamic cache models at simulation time to provide accurate results for target systems that include caches [1]. These models cause a significant simulation overhead, leading to a dramatic reduction in overall simulation performance [2], [3]. Therefore, the performance potential of SLS can only be realized under certain conditions at the current state of the art.

A potential improvement is offered by so called context-sensitive timing simulation techniques. In context-sensitive simulation, one of multiple pre-determined values is selected for each simulated execution of an instruction block. The selection is based on the current context, which abstracts the control flow leading to a particular block execution. Besides SLS, this approach has also been used in binary level simulation (BLS). In BLS the functional simulation is compiled from target binary code instead of source code and thus is significantly slower. For BLS, it has recently been demonstrated that an accurate simulation of execution time for target platforms with caches is possible without a dynamic cache model if sufficiently powerful contexts are used [4]. Transferring this approach to SLS could allow the performance limitation caused by cache modeling to be avoided. However, the BLS approach heavily relies on a tree-based context search called dynamic context selection (DCS) [5] which is significantly too slow for SLS.

As a faster alternative we propose context-sensitive timing automata (CSTA). Essentially, a CSTA encodes the behavior of a DCS based timing simulation, but only requires a table-based transition lookup instead of a tree-based context search. Consequently, the resulting simulation exhibits a high performance and accuracy even for complex embedded processors. In particular, this paper contributes CSTA based source level timing simulation, CSTA construction from context-sensitive timing data, and an experimental evaluation of this approach. This paper is organized as follows: In Section II we give an overview of SLS fundamentals and related work. The basic idea of a CSTA based simulation is presented in Section III. Section IV covers CSTA construction. Important aspects of code generation are outlined in Section V. Experimental results are presented in Section VI. Section VII concludes this paper.

II. Fundamentals and Related Work

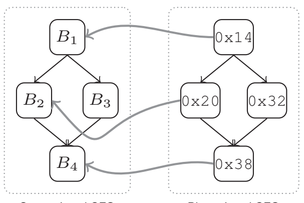
SLS, which is also referred to as host compiled simulation, has been proposed as fast, approximative technique for simulating non-functional properties of software execution on a target system [1]. Compred to instruction set simulators SLS offers improved simulation performance and retargetability. A particular focus is simulating execution time, which is divided into two subproblems in most SLS approaches: First, annotating the source code such that the target binary code execution order can be reconstructed during simulation. Second, a timing simulation is performed based on the simulated execution order. This paper is concerned with the second subproblem, but we will introduce the full approach in the following.

The first step in SLS is to establish a matching between the basic blocks in the binary and source code control flow graphs (CFG) of an application. Different heuristics are used for this purpose [6], [7]. As shown in Fig. 1a some binary level blocks might not be matched to a corresponding source level block. This can have several causes. Compiler optimizations can prevent the matching algorithm from finding a matching and complex programming constructs can prevent an insertion of function calls at the source level basic block. As shown in Fig. 1b, such missing annotations can be handled by choosing a binary path to be simulated on each execution of an annotation point. Advanced techniques for generating this path simulation code can handle complex code optimizations like loop unrolling or function inlining [8]. Furthermore, simulation performance can be enhanced by removing annotation points if the binary path can still be reconstructed [9]. Due to matching inaccuracies, the simulated execution sequence is not necessarily accurate and may even include execution orders that are impossible in the actual system.

The simulated execution sequence of target code basic blocks drives the source level timing simulation, as indicated by the `sim_timing` functions in Fig. 1b. Early approaches [10] make use of a single value to account for every execution of a particular block, while more recent approaches [8], [11] use multiple, so called context-sensitive, values for each block, which are selected at simulation time based on recent control flow. Thereby, the timing of influences such as instruction dependencies between blocks can be simulated more accurately.

An additional challenge is an approximation of the timing influence of caches. Lu et al. demonstrated that an reconstruction of accessed addresses is possible [12]. However, their results [2], [12] suggest that using these addresses to drive a dynamic cache model in SLS is prohibitively expensive. Similar results were also published by Zhao et al. [3]. As a remedy Lu et al. later proposed a fast cache simulation [2] by removing superfluous cache model accesses. For data caches the approach relies on aggregating multiple accesses for code segments such as loops, which is only possible if segments are known to be cache conflict free at annotation time. Prevalent constructs (e.g., memcpy from the C library) can not generally be assumed to be free of cache conflicts, in which case a fall back to a slow, classic dynamic model is necessary.

A potential alternative is an inclusion of the timing influence of caches in the aforementioned context-sensitive block timings. So called VIVU contexts differentiate control flow by call order and loop iteration counts and were originally developed to improve the accuracy of static timing analysis in the presence of caches [13]. For BLS, we recently demonstrated that the use of these VIVU context can enable an accurate simulation of software timing without requiring a dynamic cache model [4]. They utilize DCS [5] to select contexts in a tree-based data structure. Since the functional simulation in SLS is significantly faster than in BLS, overall SLS performance is more sensitive towards timing simulation overhead. As can be seen from the experimental results presented later in this paper, the overhead for DCS is not acceptable in SLS.



```
simulate_bb_1();
if (/*some check*/){
  simulate_bb_2();
  // do something
} else {
  // do something
}
// do something
simulate_bb_4();
```

(a) Matching

```
void simulate_bb_1(){
 /* sim preds of 0x14 */
 last_block = 1;
}
void simulate_bb_2(){
 sim_timing(0x14);
 last_block = 2;
}
void simulate_bb_4(){
 switch(last_block){
   case 1:
     sim_timing(0x14);
     sim_timing(0x32);
     break;
   case 2:
     sim_timing(0x20);
     break;
 }
 last_block = 4;
}
```

(b) Path Simulation

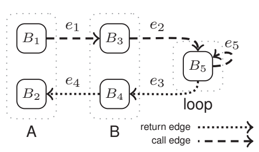
Fig. 1: Example of SLS annotation and path simulation code

III. Basic Idea

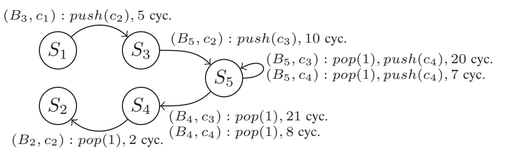
The performance overhead for the annotation and path simulation is usually low. Therefore, overall SLS performance is sensitive towards additional timing simulation overhead, which must be minimized to make a VIVU based timing simulation applicable in SLS. As a solution, we propose using a CSTA during simulation. A CSTA takes the block execution order from the path simulation as input. In contrast to other automata there is no notion of acceptance. Instead, its purpose is to track the current context and output the number of cycles to be consumed for each simulated block execution.

An example is shown in Fig. 2. The CFG in Fig. 2a contains two functions A and B. B contains a loop, which is expressed as an independent recursive routine when using VIVU contexts to enable a differentiation between loop iterations [13]. Edges can optionally represent a single call or multiple returns. For example, a return statement in a loop returns from the loop routine and the routine for the containing function. All return edges in Fig. 2a represent a single return. As can be seen in Fig. 2b, a CSTA is similar to a pushdown automaton. To support an efficient implementation, a CSTA must be deterministic and may not include ε-transitions. It has a state S_n for each basic block B_n in the CFG of a program. This state represents the most recently executed block. The automaton input is the next executing block. Transitions are selected for each input block under further consideration of the stack top, which represents the current context. A transition always includes the number of cycles by which simulated time is advanced for the most recent block execution. Furthermore, it can include a number of stack pops and one push to express context changes.

Fig. 2c shows a simulation where A calls B and two loop iterations are performed. For the first executing block B_1, the CSTA is initialized with state S_1 and the VIVU context $c_1 = \epsilon$ which encodes no control flow history. Formally this could also be expressed by a special initial state, but this state would prevent an important efficiency enhancement for the transition lookup (cf. Section V-A). When B_3 is entered (i.e., A calls B), simulation time is advanced by 5 cycles for the, now finished, execution of B_1 in the context ϵ. The VIVU context is extended



(a) Control flow graph


$(B_3, c_1) : push(c_2)$, 5 cyc.
$(B_5, c_2) : push(c_3)$, 10 cyc.
$(B_5, c_3) : pop(1), push(c_4)$, 20 cyc.
$(B_5, c_4) : pop(1), push(c_4)$, 7 cyc.
$(B_4, c_3) : pop(1)$, 21 cyc.
$(B_4, c_4) : pop(1)$, 8 cyc.
$(B_2, c_2) : pop(1)$, 2 cyc.
S_1
S_3
S_5
S_2
S_4

(b) CSTA

			Cycles	
Input	State	Stack	Consumed	Total
B_1		- uninitialized -		
B_3	S_1	c_1	5	5
B_5	S_3	c_1, c_2	10	15
B_5	S_5	c_1, c_2, c_3	20	25
B_4	S_5	c_1, c_2, c_4	8	33
B_2	S_4	c_1, c_2	2	35

(c) CSTA evaluation

Fig. 2: Basic idea of a CSTA and its evaluation

by the *context link* $(e_1, 0)$ for the call via e_1 and a recursion count of 0. The new context $c_2 = (e_1, 0)$ is pushed on the CSTA stack. When B_5 is entered, the context is extended by the context link $(e_2, 0)$ and $c_3 = (e_1, 0) \circ (e_2, 0)$ is pushed.

When B_5 is entered again in the second loop iteration, the traversed edge e_5 is considered to be recursive with respect to the VIVU context c_3, because it has the same destination as e_2 in the context link $(e_2, 0)$. When a recursive edge is traversed, a VIVU context is truncated such that it ends with the context link where the edge has the same destination. Additionally, the recursion counter of this context link is incremented. As $(e_2, 0)$ is already the last context link, no elements are truncated, leading to the context $c_4 = (e_1, 0) \circ (e_2, 1)$. Since it is not possible to return to c_3, one element is popped from the stack before c_4 is pushed. When B_4 is entered (i.e., the loop routine returns), B resumes its execution in the context c_2 it was called in. This is expressed by popping a symbol from the stack.

A finite state machine would be sufficiently powerful to express context transitions in the presented example. However, this is not true in general. If context links are removed for more complex recursions, information on the control flow history is lost in the current context. This removal is essential to ensure the set of contexts is finite. The stack is required in a CSTA to preserve this lost information. Furthermore, the granularity of VIVU contexts can be configured by two parameters. One limits the value of the recursion counters while the other limits the context length. The length limit is enforced by removing context links from the beginning of a context when a new link is appended. The example shown in Fig. 2 is also valid with a length limit of one where $c_3 = (e_2, 0)$ and $c_4 = (e_2, 1)$. In this scenario, the stack enables a return to c_2 when B_4 is entered.

IV. CSTA CONSTRUCTION

The CSTA construction takes the same input as a DCS based simulation: a control flow graph and context-sensitive timings of the binary application code. We currently apply our pre-existing methodology to obtain both [4], [5]. Two main issues must be solved: First, a set of input contexts may be incomplete in the sense that a transition between two contexts could only occur when considering multiple recently executed blocks. If the example in Fig. 2 only contained the contexts $c_2 = (e_1, 0)$ and $c_4 = (e_1, 0) \circ (e_2, 1)$, a transition pushing c_4 would not be possible without introducing nondeterminism or ε-transitions. This issue is addressed by adding so called transitional contexts. They only serve as intermediate steps to ensure all actual input contexts can be reached. For the example, the transitional context $c_3 = (e_1, 0) \circ (e_2, 0)$ would be added. Second, the input CFG may be incomplete and it is possible for the CSTA input sequence to contain block execution orders that are not reflected in the CFG. This issue is solved by additional error recovery transitions.

A. Transitional Contexts

A transitional context is a context that is added during CSTA construction to ensure an actual input context is reachable during CSTA evaluation. The basic idea for generating these transitional contexts is to ensure that for each context all contexts exist from which it could be entered in a single transition. This step is applied to all actual contexts in the input and all transitional contexts generated during CSTA construction until a fixed point is reached.

Algorithm 1 is a simplified but functionally equivalent version of the algorithm we currently use in our implementation. We adopt the context notation of Theiling [14], but further allow $a \circ b$ to not only denote a sequence of a followed by b, but also the sequence that results from a concatenation of sequences a and b. The function ADDTRANSCONTEXTS iterates over all contexts in a while loop to generate the transitional contexts for all input and generated contexts. The additional repeat loop realizes the fixed point iteration and is needed to handle special cases of recursion, which are discussed later. The context ϵ is currently required to exist for every routine in the input and is excluded from the generation of transitional contexts.

PREDS(c, C) returns the predecessors of a context. A VIVU context (except ϵ) that is entered via an edge e always ends with a context link that has the same destination as e. Thus a context $(e_1, i_1) \circ \cdots \circ (e_n, i_n)$ can only be entered by an edge e, if e_n and e have the same destination. Normal calls and recursions are differentiated in the calculation of the next context. For a normal call a VIVU context is extended by appending a new context link for the call edge with a recursion count of 0. Thus a context can only be entered by a normal call via an edge $e = e_n$ and if $i_n = 0$. The required predecessor is the context $(e_1, i_1) \circ \cdots \circ (e_{n-1}, i_{n-1})$ with the last context link removed.

As aforementioned, a call edge e is considered recursive when it has the same destination as any preceding context link edge. When a VIVU context is entered by a recursive edge, it is truncated to the context link with the edge that has the same destination as e and its recursion counter is incremented. The links lost by truncating have to be restored to calculate potential preceding contexts. For example if a call edge c is traversed in the context $(a, 0) \circ (b, 0)$ and the destination of a and c is identical, the next context is $(a, 1)$. The truncated context link $(b, 0)$ must be restored when finding the predecessors of $(a, 1)$.

Algorithm 1 Constructing transitional contexts

```
procedure AddTransContexts($C$)
    repeat
        $C_{old} \leftarrow C$
        $D \leftarrow \{\epsilon\}$
        while $C \setminus D \neq \emptyset$ do
            $c \leftarrow x \in W \setminus D$
            $D \leftarrow D \cup \{c\}$
            $C \leftarrow C \cup$ Preds$(c, C)$
        end while
    until $C \setminus C_{old} = \emptyset$
    return $C$
end procedure
procedure Preds($(e_1, i_1) \circ \cdots \circ (e_n, i_n), C$)
    $R \leftarrow \emptyset$
    for incoming call edges $e$ of destination of $e_n$ do
        if $e_n = e \wedge i_n = 0$ then
            $R \leftarrow R \cup \{(e_1, i_1) \circ \cdots \circ (e_{n-1}, i_{n-1})\}$
        end if
        for recursive call paths $d_1 \circ \cdots \circ d_m$ with $d_m = e$ do
            $p \leftarrow (e_1, i_1) \circ \cdots \circ (e_{n-1}, i_{n-1}) \circ (e_n, max(0, i_n - 1))$
            $R \leftarrow R \cup \{p \circ (d_1, 0) \circ \cdots \circ (d_{m-1}, 0)\}$
            $R \leftarrow R \cup$ RecExt$(p, d, e, C)$
        end for
    end for
    return $R$
end procedure
procedure RecExt($(e_1, i_1) \circ \cdots \circ (e_n, i_n), d_1 \circ \cdots \circ d_m, e, C$)
    $R \leftarrow \emptyset$
    for $(\hat{e}_1, \hat{i}_1) \circ \cdots \circ (\hat{e}_{\hat{n}}, \hat{i}_{\hat{n}}) \in C$ with
        $\forall l \leq min(\hat{n}, m) : \hat{e}_{\hat{n}-l} = d_{m-l}$
        $\wedge \forall l \leq min(\hat{n} - m, n) : \hat{e}_{\hat{n}-m-l} = e_{n-l}$
        $\wedge \forall l \leq min(\hat{n} - m, n) : \hat{i}_{\hat{n}-m-l} = i_{n-l}$ do
        $l \leftarrow max(0, m - \hat{n})$
        $q \leftarrow (d_1, 0) \circ \cdots \circ (d_l, 0) \circ (d_{l+1}, \hat{i}_{\hat{n}-(m-l+1)}) \circ \cdots \circ (d_m, \hat{i}_{\hat{n}})$
        $R \leftarrow R \cup \{p \circ q\}$
    end for
    return $R$
end procedure
```

The necessary restoration is based on the notion of recursive call paths $d_1 \circ \cdots \circ d_m$ which end in e. An edge is considered potentially recursive if it is part of at least one such call path. A *recursive call path* is the sequence of call edges that are not left via corresponding returns during a recursion. Currently we determine these paths by finding all elementary circuits in the CFG using Hawick's algorithm [15]. Roughly said an elementary circuit is a cycle in a graph that can not be broken into further sub-cycles. It is sufficient to only consider these elementary circuits, as all required predecessor contexts for larger cycles are already generated when processing the respective call paths of sub-cycles. Recursive call paths are calculated by removing all non-call edges and those call edges for which the circuit contains a corresponding return. Empty call paths are not considered further. This approach overestimates the set of actual call paths (e.g., when a circuit contains an invalid call/return combination). During simulation such illegal call paths can only result from matching errors and at worst result in a reduced simulation accuracy.

Preds calculates the minimal set of predecessors necessary to reach a recursive context in the case when no other contexts exist. For a context $(e_1, i_1) \circ \cdots \circ (e_n, i_n)$ when it is entered via e from a recursive call path the minimal predecessor is the context $(e_1, i_1) \circ \cdots \circ (e_{n-1}, i_{n-1}) \circ (e_n, max(0, i_n - 1))$ of the preceding recursion extended by $(d_1, 0) \circ \cdots \circ (d_{m-1}, 0)$ for the call path except for the last element $d_m = e$. The recursion counters can be set to 0, as they are removed by truncating the context when e is traversed. The simplest case is when an edge e is the only element in a call path. This is the case for the context $(e_1, 0) \circ (e_2, 1)$ from the example in Fig. 2. The recursive call path for edge e_5 is simply $d_1 = e_5$ thus $(d_1, 0) \circ \cdots \circ (d_{m-1}, 0)$ is the empty string, leading to the preceding context $(e_1, 0) \circ (e_2, 0)$.

If other contexts exist, it is possible that a different context is preferred over a generated predecessor, thereby preventing the context for which predecessors are calculated from being entered. As an example consider the context $(a, 0) \circ (b, 1)$ with a recursive call path $c \circ d$ leading to the predecessor $(a, 0) \circ (b, 0) \circ (c, 0)$. If a context $(b, 0) \circ (c, 1)$ exists in the input, DCS would use the contexts $(a, 0) \circ (b, 0)$, $(a, 0) \circ (b, 0) \circ (c, 0)$ and $(b, 0) \circ (c, 1)$. However, since the $(a, 0)$ link is dropped from the last context, it would not be possible to enter $(a, 0) \circ (b, 1)$ in a CSTA. Therefore an additional context $(a, 0) \circ (b, 0) \circ (c, 1)$ is generated. These additional contexts are generated by RecExt. To handle cases where a context for which this extension is necessary is generated later on, AddTransContexts contains the repeat loop.

B. Transition Function

The function TRANS$(q, a, w) = (p, n, u, x)$ returns the CSTA operation to be performed in state q with input a and stack top w. p is the next state. $n \in \mathbb{N}_0$ denotes the number of symbols to pop from the stack. u is a symbol to push, or $\perp$ if no push is desired. The number of cycles to be consumed is x. Pushes are performed after pops if both are included in the same transition. The successor state p is derived from the input block a. For the generated transitional contexts there is no defined cycle count n in the input. However, an appropriate value can be chosen by using the context w as input to DCS. The value of n is set to the number of returns of the edge (q, a). If this edge is a call edge a destination context has to be selected. This context is also selected using DCS, after extending the context w by the edge (q, a).

C. Error Recovery

In practice, it is very unlikely that the simulated execution order fits perfectly to a given CFG. Such problems can be caused by restrictions of the analysis used to generate the CFG (e.g., a dynamic analysis only covers observed control flow) or erroneous matchings. For SLS this leads to two issues: First, blocks not covered in a binary CFG cannot be considered during matching. This issue is beyond the scope of this publication. Second, no CFG edge (a, b) could be available for a subsequent simulated execution of blocks a and b. For a simulation using DCS accuracy can be maintained in this situation by removing elements from the end of a context until the edge of the last context link leads to the routine containing

b [4]. To transfer this approach to a CSTA based simulation, the result of TRANS(q, a, w) for all possibly missing inputs can be defined as $(a, rollback(w, a), \perp, default(q))$ where $default(q)$ is a context independent fall back cycle count for block q and $rollback(w, a)$ is the number of elements removed from w in simulation using DCS.

V. Code Generation

While the CSTA based simulation already provides a significant speed up over DCS, some aspects of code generation provide a further performance advantage. First, the transition function can be encoded by a per-state array to facilitate a fast lookup. Second, a direct expression of the CSTA in the SLS path simulation code is used.

A. Efficient Encoding of TRANS

The essential idea to efficiently encode TRANS is that in the common case (i.e., without error recovery) typically only very few inputs can occur in a given state. To allow an array-based lookup, both the context and the input block should be identified by a small number. Contexts can simply be numbered as they are added to the CSTA. Blocks must be numbered such that for each block all successors have a distinct ID. This is achieved by coloring a graph of the CFG nodes, where two nodes are connected if they share a predecessor. Currently we use a simple greedy algorithm for this purpose.

This scheme complicates the error recovery described in Section IV-C. First, a block that is not an actual successor in the input CFG may have the same ID as an actual successor leading to a successful, but wrong, transition lookup. This is handled by an additional block address verification. Furthermore a secondary look up mechanism is used for error transitions.

B. CSTA Evaluation in Path Simulation Code

To simplify switching between various simulation approaches, the CSTA evaluation was implemented in a library, where one API call is performed for each binary code block (i.e., the `sim_timing` function from Fig. 1b). However, due to the high performance of the CSTA based SLS, frequent calls cause a significant overhead. It consists of the function call overhead and the additional overhead for transition selection. During the library based simulation, transitions are selected based on the most recently executed and the subsequently executing basic blocks and the current top of stack, but during path simulation code generation the simulated execution order of binary basic blocks is already partially known. This allows to optimize the transition lookups by partially calculating the offsets in the transition tables at runtime. Generating the path simulation code and the timing simulation code in a single source file has the additional advantage that the code can benefit from further compiler optimizations, such as function inlining or interprocedural constant propagation.

VI. Experimental Results

Several programs from the Mälardalen benchmark collection [16] were used to evaluate the proposed CSTA. Benchmarks were compiled for an ARM Cortex-A9 using -O1 optimizations. For each benchmark we traced one hardware execution to generate the input to the CSTA construction using our pre-existing methodology [4]. Benchmarks executing in less than 1000 cycles where excluded, as their execution time is too low to obtain meaningful results. To limit the influence of the matching heuristic on the evaluation of the proposed CSTA base approach, benchmarks where simulated and measured instruction counts differed by more than 10% where excluded from the evaluation. This does not only cover benchmarks where the matching heuristic provides low quality results, but also benchmarks with compiler-inserted library calls for which source code is not annotated and cannot be emulated in path simulation code with sufficient accuracy.

As reference simulation we compare the proposed CSTA to SLS using DCS [4], [5]. As a CSTA encodes the same simulation logic, results should be identical to a DCS based simulation. This behavior was successfully verified by performing back-to-back tests against an implementation of DCS. Therefore we focus on simulation performance for the evaluation of the proposed approach. All simulations were compiled with -O3 and executed under Ubuntu 16.04 on a Core i5-6500 at 3.5 GHz. To provide reliable simulation runtime estimates even for short benchmarks, SLS simulations were modified to run $10,000$ iterations of each benchmark. Furthermore, 50 executions were measured of each simulation. The 10 highest and lowest execution times where removed before averaging all remaining execution times.

Fig. 3 shows the simulation throughput in million instructions per second (MIPS). With an average throughput of 2019 MIPS, the CSTA based approach provides a speed up of roughly an order of magnitude over DCS with an average throughput of 226 MIPS. Simulation performance is mainly determined by the frequency of control flow changes in a program, as they generally increase the number of block executions that must be processed by the timing simulation. The lowest performance is observed for `cover`, which loops over large switch statements where each case only increments a variable. On the other hand, `fdct` exhibits the best simulation performance, as only very little control flow changes occur compared to the amount of computations.

Fig. 4 shows the slowdown of the CSTA and DCS based

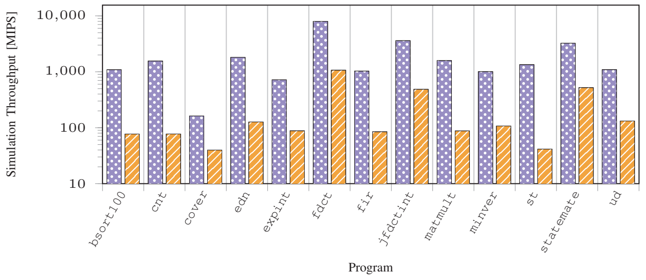


Fig. 3: Throughput of CSTA and DCS based simulations

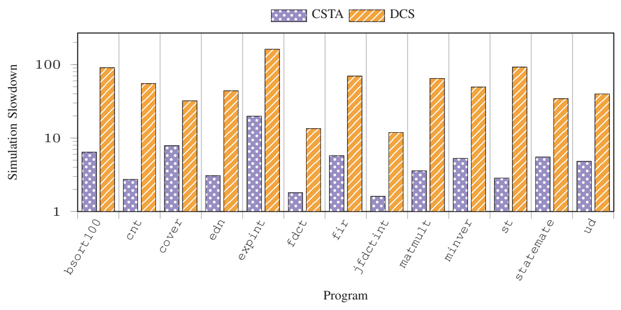


Fig. 4: Simulation slowdown compared to path simulation only execution

simulation compared to annotated code that contains a path simulation but no timing simulation. The slowdown for an ideal timing simulation that causes no overhead would be 1. The CSTA based simulation can approach this ideal in the best case, but with an average slowdown of 5.5 there is still potential for further optimizations.

For SLS using a classic (i.e., non accelerated) in-place cache simulation Lu et al. [2] report a throughput between 10 and 100 MIPS with a slowdown between factor 50 and 400 compared to an unannotated simulation. Zhao et al. [3] report a slightly better throughput of 290 MIPS for a similar cache model. These results demonstrate that in-place cache models are not appropriate in source level simulation.

For their proposed accelerated cache simulation Lu et al. [2] report an average speedup of factor 12 over the in-place cache simulation. While they provide no exact numbers, the presented graphs suggest a maximum throughput of roughly 1000 MIPS for the accelerated simulation, which is similar to the level of performance we demonstrated. However, for data caches the acceleration is only possible when memory accesses can be aggregated because they are known to be free of cache conflicts at annotation time. Otherwise a fallback to the slow in-place cache simulation is necessary in their approach. The CSTA based simulation does not have this limitation, but as a drawback does not provide cache miss statistics at the moment. We expect that cache miss statics could be approximated similarly to the cycle count using our approach.

VII. Conclusions

In this paper we proposed the use of CSTA instead of DCS to perform a source level timing simulation. Our results demonstrate that a CSTA based simulation outperforms one using DCS by roughly an order of magnitude. As a result, the application of trace-based context-sensitive timing simulation in SLS becomes feasible. In practice this means that even code running on complex embedded processors, such as the ARM Cortex-A9 used in our evaluation, can be simulated accurately and efficiently if the matching has sufficient quality.

We expect the proposed CSTA based approach to enable even further optimizations. For example, the source code annotation points could be used as automaton input if the matching is also considered during CSTA construction. Furthermore, besides the VIVU contexts we discussed in this paper, so called n-block contexts [11], [17] can also be expressed in a CSTA. As n-block contexts differentiate control flow by the n most recently executed basic blocks, they can be easily expressed by introducing multiple states per block. Thereby even a combination of VIVU and n-block contexts could be utilized in CSTA based SLS.

Acknowledgments

This work was partially supported by the BMBF project autoSWIFT under grant 16ES0358. The authors thank Florian Schneider for his implementation assistance.

References

[1] O. Bringmann, W. Ecker, A. Gerstlauer, A. Goyal, D. Mueller-Gritschneder, P. Sasidharan, and S. Singh, “The next generation of virtual prototyping: Ultra-fast yet accurate simulation of hw/sw systems,” in *Proc. Design, Automation & Test in Europe Conf. & Exhibition*, 2015.

[2] K. Lu, D. Müller-Gritschneder, and U. Schlichtmann, “Fast cache simulation for host-compiled simulation of embedded software,” in *Proc. Design, Automation & Test in Europe Conf. & Exhibition*, 2013.

[3] Z. Zhao, A. Gerstlauer, and L. K. John, “Source-level performance, energy, reliability, power and thermal (perpt) simulation,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 2016, preprint.

[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 *21st Asia South Pacific Design Automation Conf.*, 2016.

[5] S. Ottlik, S. Stattelmann, A. Viehl, W. Rosenstiel, and O. Bringmann, “Context-sensitive timing simulation of binary embedded software,” in *Proc. Int. Conf. Compilers, Architecture and Synthesis for Embedded Syst.*, 2014.

[6] S. Stattelmann, O. Bringmann, and W. Rosenstiel, “Dominator homomorphism based code matching for source-level simulation of embedded software,” in *Proc. Seventh IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign and Syst. Synthesis*, 2011.

[7] K. Lu, D. Müller-Gritschneder, and U. Schlichtmann, “Hierarchical control flow matching for source-level simulation of embedded software,” in *Int. Symp. System on Chip*, 2012.

[8] S. Stattelmann, O. Bringmann, and W. Rosenstiel, “Fast and accurate source-level simulation of software timing considering complex code optimizations,” in *Proc. 48th Design Automation Conf.*, 2011.

[9] S. Schulz and O. Bringmann, “Accelerating source-level timing simulation,” in *Proc. Design, Automation & Test in Europe Conf. & Exhibition*, 2016.

[10] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, “High-performance timing simulation of embedded software,” in *Proc. Design Automation Conf.*, 2008.

[11] S. Chakravarty, Z. Zhao, and A. Gerstlauer, “Automated, retargetable back-annotation for host compiled performance and power modeling,” in *Proc. Ninth IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign and Syst. Synthesis*, 2013.

[12] K. Lu, D. Müller-Gritschneder, and U. Schlichtmann, “Memory access reconstruction based on memory allocation mechanism for source-level simulation of embedded software,” in *18th Asia and South Pacific Design Automation Conf.*, 2013.

[13] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand, *Analysis of loops*, ser. Lecture Notes in Computer Science. Springer, 1998, vol. 1383.

[14] H. Theiling, “Control flow graphs for real-time systems analysis,” Dissertation, Universität des Saarlandes, 2002.

[15] K. A. Hawick and H. A. James, “Enumerating circuits and loops in graphs with self-arcs and multiple-arcs,” in *Proc. 2008 Int. Conf. on Foundations of Computer Science (FCS’08)*, 2008.

[16] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, “The Mälardalen WCET benchmarks – past, present and future,” in *WCET2010*, B. Lisper, Ed., 2010.

[17] R. Plyaskin and A. Herkersdorf, “Context-aware compiled simulation of out-of-order processor behavior based on atomic traces,” in *IEEE/IFIP 19th Int. Conf. VLSI and System-on-Chip*, 2011.