Accelerating Source-Level Timing Simulation

Simon Schulz, Oliver Bringmann
University of Tübingen, Sand 13, 72076 Tübingen
{simon.schulz, oliver.bringmann}@uni-tuebingen.de

***Abstract*—Source-level timing simulation (SLTS) is a promising method to overcome one major challenge in early and rapid prototyping: fast and accurate simulation of timing behavior. However, most of existing SLTS approaches are still coupled with a considerable simulation overhead. We present a method to reduce source-level timing simulation overhead by removing superfluous instrumentation based on instrumentation dependency graphs. We show in experiments, that our optimizations decrease simulation overhead significantly (up to factor 7.7), without losing accuracy. Our detailed experiments are based on benchmarks as well as real life production code, that is simulated in a virtual environment.**

I. Introduction

With the increasing complexity of modern embedded hardware systems, virtual prototypes (VPs) are becoming more and more important in industry. They are used in early development to verify functional and safety-critical properties of embedded software. However, with the increasing complexity of e.g. automotive networks, it is inevitable to test a VP's functionality in an environment as close as possible to reality.

To master this challenge, a timing simulation must be ultra-fast and still accurate. In hardware-in-the-loop setups, simulation has to be faster than real-time. A simulation technique that has gotten in focus over last years is source-level timing simulation (SLTS). In SLTS an accurate timing simulation is enabled by reconstructing target binary execution paths based on the executed source-level control flow. To find a relation between source-level and binary-level basic blocks, sophisticated matching algorithms are used. In SLTS, no single instructions are simulated, but basic blocks. This grants a tremendous simulation speed up. However, there is still a considerable simulation overhead. When simulating in host-compiled simulation environments that model complex components, overhead that can be avoided, must be avoided.

Existing SLTS approaches instrument each matched source-level basic block. In Figure 1 (b) a typical example of instrumented source code for SLTS can be seen. The `simulate_bb_x`-procedures trace source-level execution. However, e.g. `simulate_bb_4` is superfluous, since its execution can be reconstructed by elimination: if `simulate_bb_6` is called without traversing `simulate_bb_5`, it must have been executed. Thus, it can be removed without losing information during simulation. Based on this idea, we present a method to minimize SLTS overhead by identifying such superfluous procedures. Since all information is maintained, the proposed method does not sacrifice simulation accuracy.

In this work, we make the following contributions:

- A method to identify superfluous instrumentations in source-level simulation to accelerate it without loss in accuracy.
- A novel method to identify instrumentations, that can be substituted with simple flags for tracing rather than accumulate timings.
- A heuristic to contain path explosion without losing much simulation speed.
- An extensive evaluation of the introduced methods with benchmarks as well as a production code example.

The paper is structured as follows: Related work is covered in Section II. Important background concepts are introduced in Section III. In Section IV, detailed descriptions of our method, existing and novel algorithms are given. Subsequently, we present an extensive evaluation of our method in Section (V), underlining its effectiveness on benchmarks as well as real life code. We conclude our work in Section VI and present our plans for future work.

II. Related Work

Several different SLTS approaches exist. The underlying binary-to-source matching algorithms can differ [1], [2], [3], as well as the instrumentation itself [4], [5]. However, all of the given approaches have one in common: each source-level basic block that can be matched is getting instrumented as well, whether superfluous or not. To the authors' best knowledge, no research has been made to reduce instrumentation overhead in SLTS by removing superfluous instrumentation.

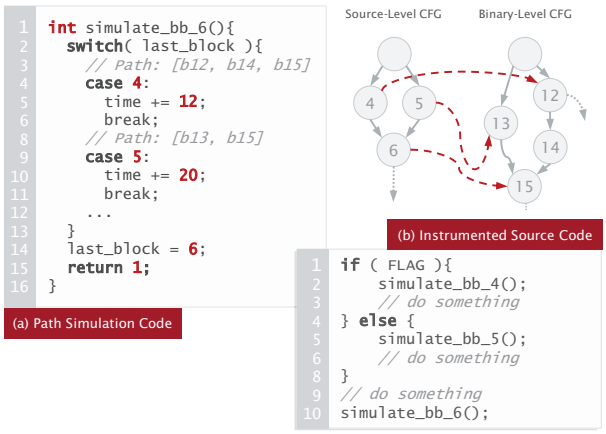
To identify superfluous procedures, we adapt techniques used in software profiling. Ball et al. present algorithms, that identify small sets of edge probes, sufficient for tracing [6]. The work is widely accepted and was mostly enhanced by dynamic optimizations for coverage testing, which are uninteresting for our work. The algorithms were applied on instrumentation point graphs for WCET analysis [7]. However, the authors focus on timing measurement. By contrast, we use these concepts in path reconstruction and to select timings based on these paths. Moreover, the authors of [6] propose an algorithm to find a minimal representation to memorize traversed edges. However, this technique only works on edges.

III. Background

A. Source-Level Timing Simulation

In SLTS, dominator-relation based matching algorithms [1], [2] allow to map source-level basic blocks (bb) to a binary-

978-3-9815370-6-2/DATE16/




```
1  int simulate_bb_6(){
2    switch( last_block ){
3      // Path: [b12, b14, b15]
4      case 4:
5        time += 12;
6        break;
7      // Path: [b13, b15]
8      case 5:
9        time += 20;
10       break;
11     ...
12   }
13   last_block = 6;
14   return 1;
15 }
```

```
1  if ( FLAG ){
2      simulate_bb_4();
3      // do something
4  } else {
5      simulate_bb_5();
6      // do something
8  }
9  // do something
10 simulate_bb_6();
```

Fig. 1. A matching, instrumented source code and path simulation code

level pendant, even for highly optimized code. For each matched pair, the source code gets instrumented to reconstruct the target binary execution path in later simulation. Thus, the sources can be compiled on a host platform and the low-level timing properties of the target still be evaluated during execution on the host. This allows to consider timings of bb transitions depending on the execution history, covering effects of caches and pipelines. Such histories are called *contexts*.

The general workflow is as follows: The source code is compiled for the target platform. On the basis of a matching algorithm and debug information, the binary's basic blocks are related to those of the source code. Thereby, conflicts and ambiguities in debug information, mostly resulting from compiler optimizations, are resolved. For each binary-level bb transition a timing is determined. Timings for bbs can be obtained using trace data, WCET analysis or other static analysis methods. A call to an instrumentation procedure is placed at each matched source-level bb. These procedures are called path simulation code and are described in detail in the following section. The resulting source code is compiled for the host and executed. Annotated timings get accumulated during execution to estimate the timing behavior of the target platform.

B. Path Simulation Code

To dynamically find the source-to-binary correlation during simulation, path simulation code is added to the source code. The instrumentation code in this paper is based on the work of Stattelmann et al. [4], using the framework described in [8]. An instrumentation procedure chooses a binary path, depending on its predecessor. This allows to consider structural differences between source-level and binary-level control flow. Depending on the choice, stored timings are accumulated during simulation.

Path simulation code and instrumented source code can be seen in Figure 1. The corresponding matching is depicted by the graphs. The presented code is simplified[1]. The procedure `simulate_bb_6` is called from the source-level bb with ID 6. It accumulates timings (Figure 1 (a), line 5 & 10), depending on the last executed instrumentation procedure (ID 4 or 5 (line 4 or 9), representing different binary paths). Therefore, it switches over the global variable `last_block` (line 2), which is set in each procedure accordingly.

[1]Usually the `last_block` variable is not the same as the procedure ID, different `last_blocks` can be set in the same instrumentation procedure.

C. Source-Level Simulation in Host-Compiled Simulation

While SLTS is meant for standalone applications only, *host-compiled simulation* in VPs extends it by models for e.g. processors, memory and caches [9]. In such environments, hardware interrupts, bus communication or peripheral participants can be modelled, which have strong impact on the simulated time. Tasks can be simulated using source-level approaches that synchronize with resource models. SLTS can be *temporally decoupled* and, with less synchronization, reduce simulation overhead [9].

Host-compiled simulation becomes particularly exciting when real communication partners are involved, e.g. by using a hardware bus-interface. In some cases, real-time simulation is required to ensure smooth communication. Thus, the execution overhead of SLTS must be reduced as far as possible, especially, when sophisticated context models are involved, such as VIVU contexts [10] or contexts by Plyaskin et al. [11]. Database approaches as described in [12] have multiple benefits for host-compiled simulation, although they are slow in execution. Other than the simulation code introduced in Section III-B, for each timing accumulation a database access is necessary and contexts must be adapted.

IV. Methodology

In this section, we introduce our method to identify and remove superfluous instrumentation procedures to reduce SLTS overhead. Our method is based on our data structure, called Instrumentation Dependency Graphs. After introducing these, we motivate and formulate a general problem description. We then present our algorithms to solve the problem and tune the results.

A. Instrumentation Dependency Graph

Starting from path simulation code (cf. Section III-B), we construct an Instrumentation Dependency Graph (IDG). An IDG of a program P is an ordered pair $IDG(P) = (V, E)$, such that V is the set of vertices and E is a set of ordered pairs of vertices of V, the edge set. Thereby, each $v_i \in V$ represents an instrumentation procedure p_i, where i is the procedure's ID (i.e. `simulate_bb_x`, x is the ID). We denote a vertex v_i is *"instrumented", iff* p_i, as well as the function call to it occurs in the final source code. By a vertex *"gets traversed in simulation"* or *"gets executed"* we mean that its corresponding procedure gets called from the instrumented source code. Edges represent dependencies between the instrumentation procedures of vertices. A directed edge $e = (v_i, v_j)$ indicates that a timing is mapped to binary path represented by the transition from p_j to p_i. Figure 2 shows an IDG and the source code that the edge (v_4, v_6) was constructed from. A path is

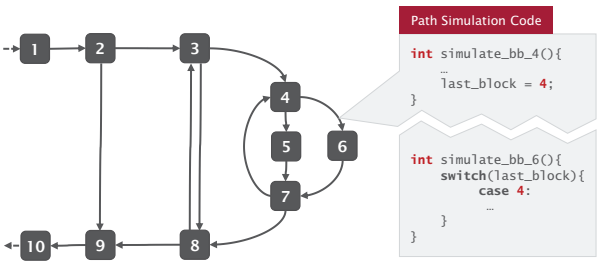


Fig. 2. Instrumentation Dependency Graph Example

a sequence of vertices which are connected to their successor in the path by an edge. It is denoted as $< v_i, v_j, v_k, ... >$.

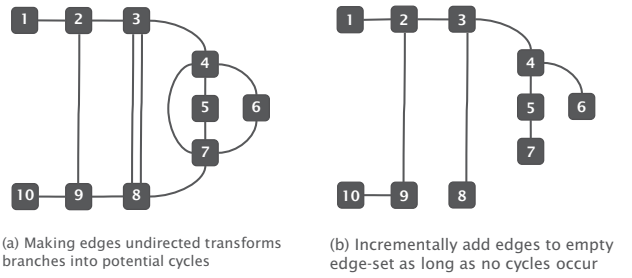
An IDG can be constructed after a matching based on intern representations or based on source code. The construction based on source code is straightforward: For each instrumentation procedure of P, a vertex is added to the graph. Afterwards, for each case in a procedure's switch statement, an edge to the predecessor is added. An IDG can but does not have to resemble the control flow on source-level.

IDGs can now be used to identify superfluous instrumentation procedures. E.g. in Figure 2, the only successor of v_1 is v_2. Hence, v_1 has not to be instrumented, because the path $< v_1, v_2 >$ can be identified, if v_2 is executed. v_3 can be reached via $< v_1, v_2, v_3 >$ or $< ..., v_8, v_3 >$. If v_8 was instrumented, each predecessor of v_3 can be reconstructed, without instrumenting v_2. Another optimization would be to remove v_5 from instrumentation. Arriving at v_7 without passing v_6 implies that v_5 was traversed. Local removal of superfluous vertices seems to be intuitive. However, finding a global, yet small set of instrumented vertices that matches the requirements, is a challenging problem. It can be generalized to the challenge: *Given an IDG, find a minimal set of instrumented vertices, such that one still can reconstruct the taken path, when reaching an instrumented vertex.*

B. Minimal Set of Instrumented Vertices

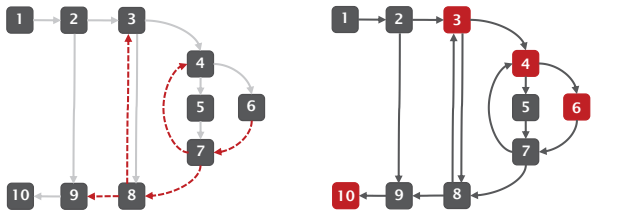
In this section, we will use IDGs combined with existing algorithms from the software testing domain, to solve the problem defined in the former section. The algorithm presented by Ball and Larus (BL) [6] can be used to find a minimal set of *edge*-probes for tracing a program, by spanning tree construction. The single steps of the algorithm applied on the graph from Figure 2, can be seen in Figure 3. A graph G is transformed into an undirected graph G^* *(a)*. This step transforms branches into potential cycles (e.g. $< v_4, v_5, v_7, v_6, v_4 >$). Afterwards, all edges are removed from G^*. Then, these edges get incrementally added to G^* again, as long as it does not contain cycles. The edges can be weighted and the insertion be ordered by these weights (cf. Section IV-E). The result is a spanning tree *(b)*. The missing edges $(E_{G^*} - E_{ST})$ need to be instrumented to solve the tracing problem *(c)*. Finding probes for tracing is equivalent to the challenge stated in the former section.

Unfortunately, edge instrumentation on source-level is non-trivial and produces additional overhead [13]. In most cases,



(a) Making edges undirected transforms branches into potential cycles

(b) Incrementally add edges to empty edge-set as long as no cycles occur



(c) Edge difference (dashed) determines edges that must be instrumented (BL)

(d) Vertices that must be instrumented identified by applying KS-BL (red)

Fig. 3. Using spanning trees to find edge instrumentation points

two probes representing start and end vertex must be instrumented to identify an edge.

Knuth and Stevenson propose a graph transformation that transforms a graph so that a corresponding edge exists for each vertex [14]. Since the transformation respects Kirchhoff's first law, the Ball-Larus algorithm can be applied and the resulting edge probes can be mapped onto the corresponding vertices to identify a small set of vertex probes for tracing. In formal terms, they define an equivalence relation between to vertices as follows: $v_i \equiv v_j$ *iff* there is vertex v_k such that $(v_k, v_i) \wedge (v_k, v_j) \in E$, or there is a finite sequence by the relation $(v_i = v_1 \equiv v_2 \equiv ... \equiv v_l = v_j)$. If the vertices of a graph G are merged following this equivalence relation, the resulting graph G' contains one edge e_{v_x} for each vertex $v_x \in V_G$. After applying BL to G', for each missing edge $e_{v_x} \in E_{G'}$ the node $v_x \in V_G$ must be instrumented.

Applying the Knuth-Stevens algorithm followed by the Ball-Larus algorithm (KS-BL, Figure 3 (d)) identifies which vertices of an IDG have to be instrumented to solve the problem defined in Section IV-A.

C. Witnesses and Accountants

In host-compiled simulation, SLTS regularly synchronizes with resource models but can also be temporarily decoupled. Thus, it is sufficient for some vertices to memorize their traversal, rather than instant synchronization. In the following, we call this type of vertices *"witnesses"*, opposed to *"accountants"*. An accountant can, but does not have to be used to synchronize, while we cannot synchronize at witnesses. One important requirement is that every possible sequence of witnesses must be finite. If simulation gets trapped in a loop and no accountant is traversed, there will be no further synchronization with the resource models.

To memorize a vertex without much overhead, each witness gets assigned an index of a bit vector. Its bit gets set, if

the vertex gets executed. An accountant accumulates timing, depending on the path represented by the set bits. Within a directed *acyclic* graph (DAG) each path can then be mapped to an unique bit-vector that identifies it.

In the following we present a method to split an IDG into several DAGs, that group witnesses. These DAGs are separated by accountants. Therefore, we deployed Algorithm 1.

Algorithm 1 Algorithm to identify witnesses

Input: instr ▷ List of instrumented vertices found by KS-BL
Input: IDG ▷ Instrumentation Dependency Graph
1: $accountants \leftarrow extractLoopEntries(instr, IDG)$
2: $witnesses \leftarrow removeAll(accountants, instr)$
3: $sortByWeight(witnesses)$
4: **for** v *in* $witnesses$ **do**
5: $G' \leftarrow removeAll(accountants, IDG)$
6: **if** $inCycle(v, G')$ **then**
7: $accountants.add(v); witnesses.remove(v)$
8: **end if**
9: **end for**
10: **return** $witnesses,\ accountants$

It gets a list of instrumented vertices (*instr*) identified by KS-BL and an IDG as input. At first, every loop entry within *instr* is added to the list *accountants* (line 1)[2]. If they are removed from the IDG, there is a high chance that only DAGs are left. However, not all loop entries have to be contained by *instr*. All other vertices in *instr* are added to the list *witnesses* and sorted as described in Section IV-E (line 2 & 3). In each iteration, namely for each witness, a graph G' is created by removing all vertices that are in the *accountants* list (line 5). If the currently inspected witness is part of a cycle in G' (line 6), it is added to the *accountants* and removed from *witnesses* (line 7). The complexity of the algorithm is $\mathcal{O}(|V|^2 \cdot |E|)$ [3].

If the vertices contained by the resulting accountants list are removed from the graph, all occurring subgraphs are DAGs. Thus, all sequences of witnesses are finite.

D. Optimized Path Simulation Code

In the example from Figure 3 (d), v_6 could be classified as a witness by Algorithm 1. Tracing indices of witnesses are assigned by incrementally giving each node a number. This is done for each DAG separately. Accordingly, the function call to p_6 is substituted by `ts_path |= 1 << 0;`, because v_6 gets assigned index 0 (`ts_path` is a global bit-vector, represented by a 64 bit variable). The optimized instrumentation is given in Figure 4. If coming from v_4 itself, the path variable is examined and the path can be reconstructed (lines 6-14). The path timings can be determined by walking over all paths between accountants during optimization, accumulating timings and memorizing the resulting bit-vector. In the examples in Section V, 64 bit where enough to uniquely address each witness within one DAG.

[2]Loop entries can be identified by finding back edges in a graph [15].

[3]A DFS that searches for cycles runs in $\mathcal{O}(|V| + |E|)$. Removing nodes from G is possible in $\mathcal{O}(|E|)$ in adjancent list based graphs [16]. Creating G' is in $\mathcal{O}(|V| \cdot |E|)$. This has to be done for each potential witness.

```
1  int simulate_bb_4(){
2    switch( last_block ){
3      case 3:
4          ...
5      case 4:
6          switch( ts_path ){
8              case 0:
9              // path <4,5,7,4>
10                 time += ...
11                 break;
12             case 1:
13             // path <4,6,7,4>
14                 time += ...
15                 break;
16         } break;
17         ts_path = 0;  // reset }
```

(a) Path Simulation Code

```
1  simulate_bb_4();
2  if ( FLAG ){
3    // former simulate_bb_6
4    ts_path |= 1 << 0;
5    ...
6  } else {
8    //simulate_bb_5 removed
9    ...
10 }
```

(b) Instrumented Source Code

Fig. 4. Optimized path simulation code including a global path bit-vector

E. Vertex Weights

The spanning tree construction as well as the order in which witnesses are transformed to accountants depends on a sorting based on vertex weights. Vertices added last in the spanning tree construction are more likely to be instrumented. To achieve a high number of witnesses using Algorithm 1, we want loop entries to be accountants. Further, in simulation, inner-loop vertices are more often executed than others and produce more overhead, thus, we assign lower weights. This leads to the following weighting function for a vertex v:

$$weight(v) = 3 \cdot isLoopEntry(v) - loopLevel(v)$$

where $isLoopEntry$ returns 1 *iff* a node v is a loop entry, 0 else. Multiple tests showed that 3 is an appropriate coefficient since it forms a dominance trade-off of loop-entries over nested nodes. For KS-BL the candidate list is sorted in ascending order, while for Algorithm 1 it is sorted in descending order, using these weights.

F. Further Optimization

An algorithm to further optimize the results of KS-BL exists [17], [18]. The heuristic, finding Locally Minimal Solutions (LMS), checks if a node has no edge from its predecessors to its successors. If so, that vertex must not be instrumented. We apply LMS on the DAGs we receive after Algorithm 1 (namely, skipping accountants). Various tests showed, that this stage is best, so that LMS does not interfere with the weighting from the spanning tree construction.

G. Path Reduction

The number of paths between accountants can grow very fast. For each of these paths, a pre-calculated timing must be stored, no matter if in source code or a database. When talking about *additional* paths, edges between accountants are ignored. To reduce the number of additional paths and thus, data, we deployed Algorithm 2. It takes a list of accountants and witnesses as identified by KS-BL + LMS, the IDG and a threshold limiting the number of additional paths. Accountants are removed from the IDG, returning a list of DAGs *DAGList* (line 1). The procedure *countPaths* returns a sum of all additional paths of all DAGs. While this number is bigger than the threshold, the following iteration takes place (line 3-12): The DAG with the highest number of additional paths is

Algorithm 2 Algorithm to reduce the number of paths

Input: accountants ▷ List of accountants
Input: witnesses ▷ List of witnesses
Input: IDG ▷ Instruction Dependency Graph of the program
Input: threshold ▷ Threshold limiting number of additional paths

1: $DAGList \leftarrow removeAll(accountants, IDG)$
2: $numPaths \leftarrow countPaths(DAGList)$
3: **while** $numPaths > threshold$ **do**
4: $maxDAG \leftarrow findMaxPathsDAG(DAGList)$
5: $v = maxEdgeWitness(maxDAG, witnesses)$
6: **if** $v == null$ **then**
7: **break**
8: **end if**
9: $accountants.add(v);\ witnesses.remove(v)$
10: $DAGList \leftarrow removeAll(accountants, IDG)$
11: $numPaths \leftarrow countPaths(DAGs)$
12: **end while**
13: **return** $witnesses,\ accountants$

Table I. Accuracy of unoptimized and optimized SLTS results

Benchmark	native ns	unopt ns	opt ns
cnt	120012	109120	109120
matmult	4148538	4084842	4084842
statemate	20448	17521	17521
edn	1598790	1814904	1814904
crc	1007764	899211	899211

returned by *findMaxPathsDAG* (line 4). From this DAG, the witness with the highest number of edges is extracted (line 5). The idea is that these vertices maximally reduce the amount of additional paths in the DAGs, if moved from witnesses to accountants (line 9). Afterwards, *DAGList* and *numPaths* get updated (line 10 & 11). This process is repeated until the number of additional paths is under the threshold or no more witnesses can be converted. The complexity of the algorithm is $\mathcal{O}(|V|^2 \cdot |E|)$ [4].

V. Results

In this section, we will evaluate the effect of the proposed reduction on the overhead of SLTS. For evaluation we used applications from the Mälardalen Benchmark [19]. *cnt, edn* and *matmult* are classical matrix and vector based applications, while *statemate* and *crc* are applications with a more interesting control flow. We compiled these applications for an embedded evaluation board containing an $SPC56EC64$ (PowerPC), using *Wind River Diab*. We matched the applications' source code with the resulting binary. The timings where extracted from a trace (using *iSystems BlueBox*). The simulated time of an unoptimized and an optimized instrumented source-code (BL-KS & LMS) compared to the native execution time on the micro-controller can be seen in Table I. *native* is the execution time on the micro-controller, *unopt* the simulated time of the unoptimized and *opt* of the optimized instrumentation. Our optimization did not influence accuracy.

[4]The number of paths in a topologically sorted DAG can be determined in $\mathcal{O}(|V|+|E|)$ (it must be sorted once in $\mathcal{O}(|V|+|E|)$) [6]. Removing nodes from G is possible in $\mathcal{O}(|E|)$ in adjancent list based graphs [16]. Creating *DAGList* runs in $\mathcal{O}(|E| \cdot |V|)$, in worst-case, for each witness (line 6 & 7).

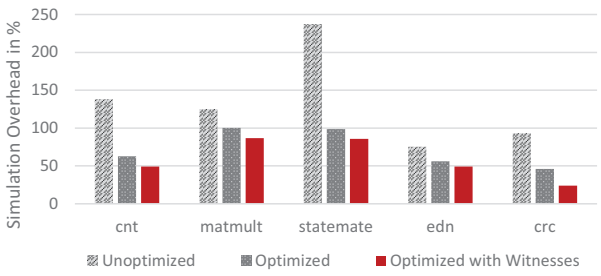


Fig. 5. Simulation overhead with unoptimized and optimized instrumentation (with and w/o witnesses)

A. Benchmark Evaluation

In Table II, the detailed results of the reduction algorithm can be seen. We split the table in two parts: *unoptimized* and *optimized* instrumentations. *#v* represents the number of instrumentation procedures, while *#v calls* represents how often these procedures are called during execution. In the optimized columns *#acc* stands for the number of accountants and *#wit* for the number of witnesses in the source code. Calls are given, correspondingly.

The number of calls to instrumentation procedures *#v calls*, compared to calls to accountants was reduced by at least 26% (edn) and at most by 79% (statemate). LMS seems to not have a big impact on the results. The number of additional paths as defined in Section IV-G, is given in the last column. We set the threshold in Algorithm 2 to 1000. If the paths initially (after BL-KS, LMS) exceeded 1000, their number is denoted in brackets. statemate was the only benchmarks with a remarkably high number of additional paths between accountants. This is, because statemate implements a state machine with a lot of consecutive branches without iteration in between. Path reduction transformed 7 witnesses to accountants to reduce paths from 65781725201 to 521.

Figure 5 depicts the simulation overhead caused by instrumentation code as introduced in Section III-B[5]. It can be seen, that simulation overhead can be strongly decreased. Especially crc that contains a high share of witnesses benefits from our method (reduced by a factor of 3.9). Note that the simulation overhead stands in relation with the actual, functional computation. As already mentioned, the used original instrumentation code is the simplest possible when evaluating bb transitions. Thus, the evaluated overhead can be seen as a baseline.

B. CAN Interface of AUTOSAR Stack

We applied our algorithms to a CAN-interface production code from an AUTOSAR stack. The source code contains about 3.1k lines of code. From initially 532 instrumented vertices, 158 are left as accountants and 144 as witnesses after optimization. Paths could be reduced from 5509 to 907 by adding only two witnesses to accountants. We evaluated 3 procedures in a host-compiled simulation: `rxIndicate` indicates when a CAN message was received, `transmit` triggers a

[5]We measured 1000 executions of the uninstrumented source code and compared it to 1000 executions of the both variants. The overhead (%) is calculated by: $((t_{sim} - t_{native})/t_{native} * 100$

Table II. Results of instrumentation procedure reduction

	unoptimized		optimized						
					without LMS		with LMS		
Benchmark	#v	#v calls	#acc	#acc calls	#wit	#wit calls	#wit	#wit calls	paths
cnt	21	550	6	221	2	62	1	61	19
matmult	22	18533	7	9262	2	401	1	400	20
statemate	208	83	14	18	119	10	114	9	521 (65781725201)
edn	43	4075	14	3232	0	0	0	0	44
crc	28	7432	6	2388	8	1192	6	1109	94

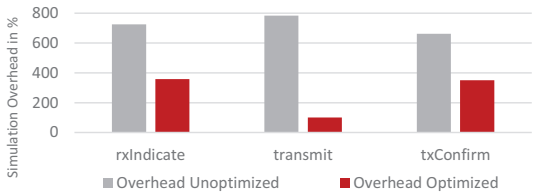


Fig. 6. Simulation overhead in a host-compiled simulation environment

transmission, while `txConfirm` confirms the transmission. The simulation was triggered by a real micro-controller using a hardware CAN interface. Each accountant calls a (very) small processor resource model, which increases the simulation overhead. The results can be seen in Figure 6. The simulation overhead was significantly reduced (from ~770% to ~100% in `transmit`). The optimization of this largest of the experiments took about 2.8 seconds on an Intel i5-3470 CPU with 3.20GHz and 8GB Ram (statemate took ~1.6 seconds), using a Java implementation.

VI. Conclusion

In this paper, we presented a novel method for removing superfluous instrumentation procedures from SLTSs to reduce overhead. Furthermore, we offered an algorithm to classify these vertices into witnesses and accountants for additional speed-up. A path reduction heuristic handles the problem of path explosion and the size of incremental data. We showed in experiments, that the simulation overhead can be reduced tremendously using our reduction, in benchmarks (up to a factor of 3.9) as well as in production code (up to a factor of 7.7). Experiments with real life source code examples, executed in a host-compiled simulation, showed how effective our method works when resource models are involved.

The methodology is lossless in means of dynamic path reconstruction. Execution paths can be reconstructed at every point dynamically. This qualifies it for more complex simulations e.g. involving contexts [12], [11] or databases. Therefore, only timing (pre-) accumulation must be modified. Pre-accumulating paths can increase accuracy, due to history dependent timings. Accumulating each possible path in advance would yield fantastic results, however, the resulting data would be too large. The path reduction heuristic offers an adjustable way, to decrease paths efficiently.

Especially when heading towards complex host-compiled simulations that involve large timing databases [12] combined with complex context models, our reduction will be essential. In hardware-in-the-loop simulations, the simulation speed must exceed real-time. However, even in simple simulations there is no real drawback in using our method.

In future, we will investigate host-compiled simulation and adapt our method towards the specific needs. This is e.g. the identification of hotspots for interrupts and the modification of witness classification, accordingly. Furthermore, we will have a look at the optimization of context based simulations. Another topic of interest is the integration of simulations with explicit cache simulation.

References

[1] S. Stattelmann, O. Bringmann *et al.*, "Dominator homomorphism based code matching for source-level simulation of embedded software," in *CODES+ISSS*, 2011.

[2] K. Lu, D. Müller-Gritschneder, and U. Schlichtmann, "Hierarchical control flow matching for source-level simulation of embedded software," in *System on Chip (SoC), 2012 International Symposium on*.

[3] D. Mueller-Gritschneder, K. Lu *et al.*, "Control-flow-driven source level timing annotation for embedded software models on transaction level," in *14th Euromicro Conference on Digital System Design (DSD)*, 2011.

[4] S. Stattelmann, O. Bringmann *et al.*, "Fast and accurate source-level simulation of software timing considering complex code optimizations," in *DAC*, 2011.

[5] Z. Wang and J. Henkel, "Accurate source-level simulation of embedded software with respect to compiler optimizations," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2012.

[6] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *ACM Transactions on Programming Languages and Systems*, 1994.

[7] A. Betts and G. Bernat, "Tree-based wcet analysis on instrumentation point graphs," in *ISORC*, 2006.

[8] C. Gerum, O. Bringmann, and W. Rosenstiel, "Source level performance simulation of gpu cores," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, 2015.

[9] O. Bringmann, W. Ecker *et al.*, "The next generation of virtual prototyping: Ultra-fast yet accurate simulation of hw/sw systems," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. EDA Consortium, 2015.

[10] H. Theiling, "Control flow graphs for real-time systems analysis," *Universität des Saarlandes, Diss*, 2002.

[11] R. Plyaskin and A. Herkersdorf, "Context-aware compiled simulation of out-of-order processor behavior based on atomic traces," in *VLSI and System-on-Chip (VLSI-SoC), 2011 IEEE/IFIP*. IEEE, 2011.

[12] S. Ottlik, S. Stattelmann *et al.*, "Context-sensitive timing simulation of binary embedded software," in *CASES '14*, 2014.

[13] R. L. Probert, "Optimal insertion of software probes in well-delimited programs," *Software Engineering, IEEE Transactions on*, no. 1, 1982.

[14] D. E. Knuth and F. R. Stevenson, "Optimal measurement points for program frequency counts," *BIT Numerical Mathematics*, vol. 13, 1973.

[15] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques*. Addison wesley, 1986.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms second edition," *The Knuth-Morris-Pratt Algorithm*, 2001.

[17] D. Baca, "Tracing with a minimal number of probes," in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*. IEEE, 2013.

[18] ——, "Software probe minimization," 2014, uS Patent 8,762,960.

[19] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen wcet benchmarks: Past, present and future." *WCET*, vol. 15, 2010.