# Hybrid Source-Level Simulation of Data Caches Using Abstract Cache Models

Stefan Stattelmann[†] Gernot Gebhard[‡] Christoph Cullmann[‡] Oliver Bringmann[†] Wolfgang Rosenstiel[†§]

[†]FZI Forschungszentrum Informatik
Haid-und-Neu-Str. 10–14
D-76131 Karlsruhe, Germany

[‡]AbsInt Angewandte Informatik GmbH
Science Park 1
D-66123 Saarbruecken, Germany

[§]University of Tuebingen
Sand 13
D-72076 Tuebingen, Germany

***Abstract*—This paper presents a hybrid cache analysis for the simulation-based evaluation of data caches in embedded systems. The proposed technique uses static analyses at the machine code level to obtain information about the control flow of a program and the memory accesses contained in it. Using the result of these analyses, a high-speed source-level simulation model is generated from the source code of the application, enabling a fast and accurate evaluation of its data cache behavior. As memory accesses are obtained from the binary-level control flow, which is simulated in parallel to the original functionality of the software, even complex compiler optimizations can be modeled accurately. Experimental results show that the presented source-level approach estimates the cache behavior of a program within the same level of accuracy as established techniques working at the machine code level.**

***Index Terms*—System analysis and design; Timing; Modeling; Software performance; Cache memories;**

## I. Introduction

The growing complexity of embedded systems and the ever-increasing use of MPSoCs (multiprocessor system-on-chip) and multi-level memory hierarchies makes estimating non-functional properties like execution time and power consumption a challenging task. As these properties are determined by the interaction of hardware and software components, neither of them can be considered in isolation. Instead, their interdependency must be modeled in a fine-grained fashion to obtain accurate results. Caches can heavily influence the non-functional properties of an embedded system and are a prime example of how software can influence these properties. The reason for this is that the cache contents are almost solely determined by the executed machine instructions and the way these instructions access memory cells.

The effects of caches on the efficiency of an entire system can be tremendous. On the one hand, a memory access missing the cache takes significantly more time as it causes an additional access to main memory. Hence, the number of cache misses significantly affects the execution time of the software and thereby the responsiveness of the system. On the other hand, the difference in energy consumption between those memory accesses hitting the cache and those which must go all the way through the memory hierarchy to main memory can be huge. Therefore caches are also very important for the power analysis of embedded systems.

Additionally, not all types of caches are alike. The possible access patterns to the instruction memory can be derived from the structure of the machine code. Therefore analyzing the behavior of instruction caches is relatively easy. Various solutions for the analysis of instruction caches exist in the literature. On the other hand, estimating the cache utilization of data caches and unified caches, which store instructions and data, is more challenging. Data accesses are highly dependent on the input of the executed program and its execution history. Thus, a static analysis of the data cache behavior is often not completely accurate, as it has to abstract the program state to cover all possible program runs. In contrast to this, simulation-based approaches do not have this limitation, as they model the cache behavior of a single program run only. Yet accurately modeling the data cache in a binary-level simulation usually comes with a large performance loss.

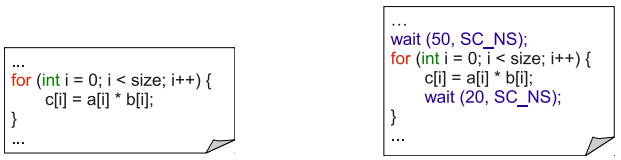
This paper proposes a hybrid cache analysis which combines static analyses of binary-level control flow and memory accesses in the machine code with a high-speed source-level simulation of application code. The destination of memory accesses is determined using binary-level interval analysis of processor registers. This approach allows a fast and precise evaluation of the data cache behavior as the potential destination of every memory access in the machine code can be determined. In contrast to previous publications in the area of source-level cache simulation, the presented technique is able to track all memory accesses present in the actual binary code. There is neither a restriction to global variables nor the implicit assumption that memory accesses in the host-compiled application code can be translated to memory accesses in the target architecture. Thus, the presented approach is able to cover the actual cache behavior of the application software running on the target architecture more accurately.

The remainder of this work is organized as follows: the next section gives an overview of existing methods for timing and cache analysis of embedded software. Section III introduces the concept of abstract caches known from static cache analysis which will be adapted for the presented hybrid approach. In Section IV, the proposed technique for source-level simulation of data caches will be described in detail. Section V provides an experimental evaluation of the proposed technique. Finally, the last section gives a summary of the presented work.

## II. State of the Art and Related Work

For a given processor architecture, properties of software components can be evaluated using an instruction set simulator (ISS) modeling the functionality of the target processor. In state-of-the art binary simulators like Synopsys CoMET [1] or Imperas Open Virtual Platforms [2], just-in-time (JIT) compilation is used to translate instructions of the target processors to instructions for the simulation host. Non-functional properties like the cache behavior can be determined using online cache models integrated into the ISS or by generating traces for an offline model like Dinero IV [3]. As the dynamic translation of binary code and the simulation of non-functional properties introduces a significant overhead, an ISS-based execution of

978-3-9810801-8-6/DATE12/©2012 EDAA




(a) Original Source Code (b) Annotated Source Code

Fig. 1: Simple Timing Instrumentation

software is often considerably slower than the native execution of the same program on the simulation host.

As an alternative to simulation-based approaches, the execution time and cache behavior of embedded software can be determined using static analysis. This is achieved by safely approximating all possible executions of a program. The academic framework OTAWA [4] and the commercial tool AbsInt aiT [5] use abstract interpretation [6] of machine instructions to determine a worst-case execution time (WCET) bound of a program. The computed WCET bound is an upper bound for the execution time of all possible executions independent of the program inputs. Internally, the WCET analysis uses a model of the processor pipeline to determine the execution time of basic blocks, and abstract caches [7] to determine the delay of memory accesses in the binary code. As the state space for the pipeline and the cache states can grow very large for sophisticated target architectures, these analyses must use (safe) approximations for the potential states of the analyzed system. Additionally, only the worst-case program path for the analyzed property is computed. If a program is tested with a fixed set of inputs, for instance during architecture exploration in an early design phase, the results for the worst-case path can significantly deviate from those for the tested program path. Usually system designers are interested in average case execution behavior and therefore static analysis is not the first choice for such use cases.

Determining non-functional properties of embedded software through source-level simulation has been proposed as another alternative for the use of an ISS. In a source-level simulation, the source code of software components is enriched with annotations describing the non-functional properties to be analyzed. The annotated source is then compiled for the simulation host. Using the resulting host-compiled binary, non-functional properties of the software running on the target architecture can be obtained while executing the software on the simulation host.

If there is a unique relation between basic blocks[1] in the source code and the binary code, the desired properties can be directly annotated to the source code. This is demonstrated for a simple program in Figure 1, where the execution time of binary-level basic blocks is represented by calls to the function `wait`. In case compiler-optimizations are used when creating the machine code for the target architecture, the structure of the source code and the final machine code can be completely different. Hence, adding source-level annotations for a given basic block in the machine code can become very complicated. Therefore, source-level simulation was originally limited to programs compiled without compiler optimizations [8], [9] or only with optimizations that preserve [10], [11] or simplify [12] the structure of the program. Additionally, it cannot be determined from the source code whether a read or write access at the source-code level, e.g. an access to a variable, is translated to a real memory access, meaning a load or store instruction. The reason for this is that it depends on the compiler whether the value of a source-level variable is stored in main memory or in a register. Moreover, the storage location of a variable often changes during its lifetime. This is a large obstacle for an accurate estimation of the data cache behavior in a source-level simulation. Thus, the aforementioned approaches can only model the instruction cache, if the cache behavior is considered at all.

In order to add annotations for source-level simulation even if the compiler performs optimizations which change the structure of a program, the approaches presented in [13], [14], [15] use a modified compiler to emit timing-annotated source code. To some extend, this approach can also be used to annotate memory addresses using a memory address translation from host addresses to target addresses [16]. A similar translation scheme not requiring a modified compiler has been proposed in [17]. Instead of using the compiler to emit memory accesses, the compiler-generated debug information is used to translate accesses to source-level variables to memory accesses in the target architecture. Both of these approaches implicitly assume a tight coupling between memory accesses on the simulation host and the target architecture. Furthermore, they do not provide any guarantees that an actual memory access in the machine code is represented in the simulation. In contrast to these existing approaches, this work will present a method to model memory accesses in a source-level simulation without relying on memory access patterns on the simulation host.

## III. Static Cache Analysis

The basic idea behind a static cache analysis is to classify memory accesses in the program as a cache hit or a cache miss. As these classifications must hold for all possible executions of a program, a precise classification for every access is not always possible. Consequently, additional classifications besides hit or miss are used, for instance a miss for the first access to a memory cell only. For some accesses, a classification might not be possible at all. In this case, further analyses which use the result of the cache analysis, like a pipeline analysis for determining the WCET of a basic block, must either use a worst-case assumption or model both possibilities. For space reasons, the concepts of static cache analysis will only be sketched in the following paragraphs. A more detailed description of static cache analysis can be found in [7].

The example in Figure 2 illustrates one of the reasons why not every memory access in a program can be classified as cache hit or cache miss without executing the program. In this simple example, the control flow graph (CFG) of a program with one memory access in every basic block is shown. At the outgoing edge of each basic block, the cache state is shown after the memory access in the respective basic block has been executed. The cache is assumed to be a fully associative cache with three cache lines and the least recently used (LRU) replacement policy. After every memory access, the most recently used cache entry is depicted at the top of the cache while the least recently used entry is located at the bottom. The cache state after the CFG nodes 1–3 contains all memory cells that were accessed along the respective program path from the start node. The cache state at node 4 cannot be determined accurately by static analysis, as it depends on the path through which the node was reached. Depending on which part of the branch at node 1 is taken, either memory cell **B** or memory cell **C** would be the most recently used entry in the cache before the execution of node 4. For the LRU cache in this example, the memory cell **D** is always the most recently used cache entry after the execution of node 4 and **A** is always the least recently used entry.

[1] A basic block is a maximal sequence of consecutive source-level statements or machine instructions with no possibility of branching inside the basic block.

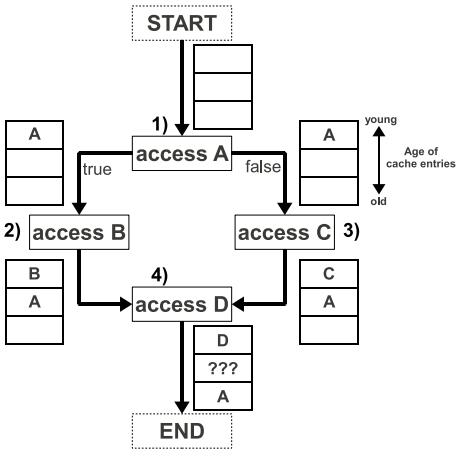
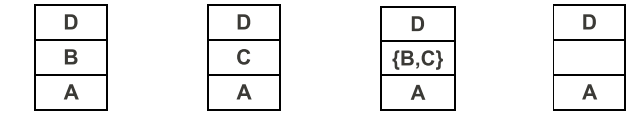



Fig. 2: Cache Analysis Example




(a) $1 \rightarrow 2 \rightarrow 4$ (b) $1 \rightarrow 3 \rightarrow 4$ (c) May Cache (d) Must Cache

Fig. 3: Cache States

To model all possible cache states accurately, a static cache analysis uses the concept of abstract cache states determined by a must and a may analysis. The must cache analysis determines all entries which are guaranteed to be in the cache. For the example in Figure 2, this holds for entries **A** and **D**. Accesses to these memory cells directly after node 4 are guaranteed to be cache hits. On the other hand, the may cache analysis determines all memory cells which could potentially be in the cache. In effect, all other entries are guaranteed cache misses.

The various cache states after the execution of node 4 from the cache analysis example are show in Figure 3. If the concrete path through the program is known, the concrete state of the cache can be determined precisely. The may and must caches are an abstraction for *all* concrete states. In this simple example, the may and the must cache can be determined by the union and intersection of the entries in every cache line for every concrete state. The concept of abstract cache states can also be applied if the precise destination of a memory access cannot be determined. Instead, it suffices to determine the possible address range of a memory access to estimate the potential cache state. These ranges can be determined using interval analysis [6] of processor registers at the machine code level [18]. A memory access approximated by an address interval may affect multiple cache sets in one update of the abstract cache state. In the worst case, such an abstract memory access affects all abstract cache sets. In contrast to discarding the whole abstract cache state, only little information is lost.

Figure 4 illustrates how interval analysis can be used to estimate memory accesses using a simple sequence of ARM instructions. The first instruction in the example loads data from the address stored in register `r2` and stores it in register `r1`. Thus, the value of `r1` after executing the first instruction solely depends on the value of the respective memory cell. Without further knowledge about the data sections of the program and previously executed instructions, the possible value range of `r1` covers all values of a 32-bit register. The second operation executes a logical right shift by 24 positions. Therefore the upper 24 bits of `r1` are guaranteed to be zero after this operations,

```
...
0x8000 LDR   r1, [r2]
0x8004 LSR   r1, r1, #24
0x8008 LDR   r3, [r1]
...
```

Fig. 4: Example Machine Code for Interval Analysis

reducing the possible value range to the interval `[63..0]`. Using this information, the memory area accessed by the third instruction can be narrowed down to the value range of `r1`. Similarly, the contents of the data cache which might be influenced by this access can also be determined. As a result, the cache behavior of a program can be estimated by analyzing all possible cache states for all program paths using abstract caches.

This work proposes the adoption of abstract cache modeling from static cache analysis for the source-level simulation of the cache behavior of a program. Using explicit enumeration of all possible cache states among all possible execution paths through a program is not feasible in general. Yet this analysis technique allows for a compact representation of cache states, from which simulation-based approaches can also benefit.

## IV. Proposed Dynamic Estimation Technique

As stated in the introductory sections, compiler optimizations disrupt annotations of machine code properties to the source code of a program. To overcome the issue of matching the structure of the source code and the machine code, this paper proposes the concept of binary-level path simulation as described in [19] and [20]. Furthermore, a hybrid approach to simulate caches is introduced which combines a static analysis of memory accesses in the machine code with a source-level simulation of abstract cache states. As a result, accurate estimates of the data cache behavior can be obtained during a source-level simulation. Figure 6 depicts the complete annotation work flow which will be explained below.

### A. Relating Source Code and Machine Instructions

If it is undesirable or infeasible to use a modified compiler, annotating attributes of the machine code to the source code of a program is usually done based on the line references in the compiler-generated debug information. This information can be generated by most compilers, but it is not guaranteed to be exact if compiler optimizations are used. This issue is illustrated by the code snippets in Figure 5 using *Loop Invariant Code Motion* as an example compiler optimization: the expression `c*d`, which is constant for all iterations of the loop, is moved in front of the loop. As the expression originated from the loop body, the debug information for the binary code will map it to the loop body in the source code, although the respective machine instructions are no longer part of the loop. Hence, the relation between source code and binary code is no longer correct with respect to the execution order of the program statements. To allow a precise source-level simulation of binary code execution, these inconsistencies in the compiler-generated debug information must be eliminated.

Contradictory data contained in the line references can be detected using the *dominator relation* of nodes in the source-level and binary-level control flow graphs[2]. After compiling a program for the target architecture in step 1 of Figure 6, this relation can be calculated for the source-level and binary-level control flow graph. The line references are extracted from the

[2]A node in a control flow graph *dominates* another node if every path from the entry node to the second node also includes the first node.

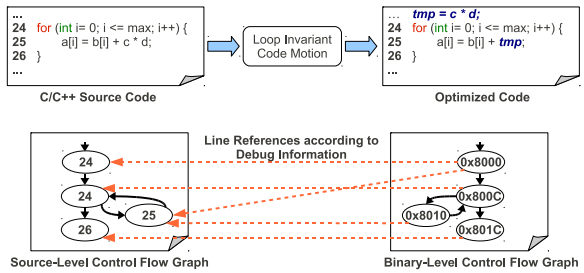



Fig. 5: Effect of Optimizations on Debug Information

machine code in step 3. They are used to derive a mapping between both control flow graphs. To detect and remove invalid line references, the mapping is reduced to entries preserving the dominator relation on both levels by step 5. In the resulting mapping, the dominator relation of a pair of nodes in the binary-level CFG and the nodes to which they are mapped in the source code must be identical. Accordingly, if one binary-level basic block is always executed before a second one, the same relation holds for their respective source-level entries in the mapping. With this reduced mapping, more accurate annotations can be added to the source code. For space reasons, a complete description of this approach cannot be presented. Please refer to [20].

### B. Simulating Binary Control Flow

Even if incorrect debug information can be detected, there is not necessarily a unique source code location for every binary-level basic block. For instance if a function is inlined by the compiler at multiple call sites of the function, the respective basic blocks in the machine code will all reference the same source code location. Yet, if the predecessors of the respective basic blocks are known, it is usually clear which instance in the binary code would be executed during an execution on the target processor.

To overcome this ambiguity in the mapping between source code and binary code, annotations will be selected dynamically during simulation. Based on the corrected line references, the binary-level control flow is analyzed to create code for performing a dynamic reconstruction of executed basic blocks (Figure 6, step 9). For source code locations which are referenced by multiple binary-level basic blocks, the generated path simulation code decides which basic block would be executed during an execution of the actual target binary. Hence knowledge about the execution history (e.g. the call stack) can be used to select the correct annotations for every basic block during the host-compiled execution of the instrumented source code.

The concept of dynamic path simulation has several advantages. First of all, it makes the annotation framework more robust against incorrect entries in the debug information. As the dynamic path reconstruction only simulates feasible paths through the binary-level CFG, incorrect annotations are simply ignored. Furthermore, changes of the program structure during compilation can be simulated accurately. It has already been shown in [19] that this concept allows modeling the effects of function inlining and partial loop unrolling. Especially the latter is hardly possible with static annotations.

### C. Detecting Memory Accesses

An exact simulation of the data cache behavior of a program requires precise knowledge about the actual memory accesses *on the target processor*. As the compiler is free to choose whether a value is stored in memory or in a register, there is no guarantee that an access to a source-level variable triggers a

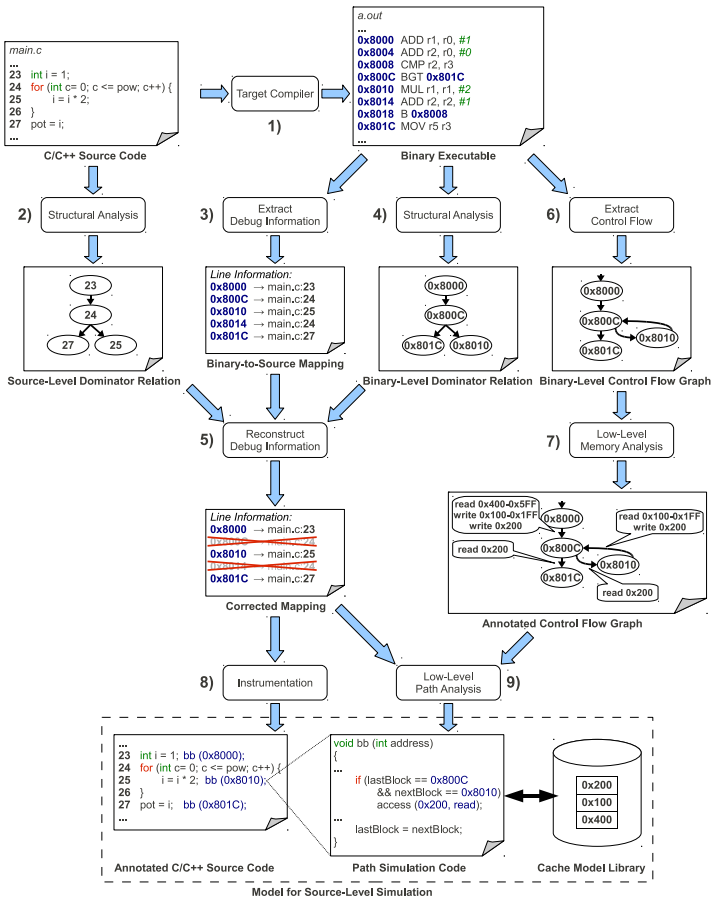



Fig. 6: Proposed Memory Annotation Work Flow

memory access in the machine code. To some extent, compiler-generated debug information can help in translating variable names to addresses of the target architecture, as proposed in [17]. In particular local variables are subject to repeated relocation during optimized compilation, as values might move from main memory to a register and vice versa. Hence, it is very likely that a translation of variable names or host addresses to target addresses does not cover all memory accesses in the machine code.

The proposed alternative is to annotate memory accesses in a similar fashion as it has been done with the execution time of basic blocks in previous work. It is often possible to determine value ranges for registers at the machine code level using interval analysis. With this information, the memory areas accessed by a memory instruction can be narrowed down. Therefore, memory accesses executed within a basic block can be annotated just like any other static property of the machine code. In the presented annotation work flow, the information about the memory areas accessed by a given binary-level basic block is obtained using abstract interpretation. Initially, the possible value range of every processor register is determined using interval analysis. Subsequently, the addresses accessed by every memory instruction can be derived using these intervals and the semantics of the instruction set for the target processor. This results in an annotated control flow graph containing range information for the destination of every memory instruction, cf. step 7 of Figure 6.

Based on the annotated CFG, memory access annotations can be generated for source-level simulation. Using this information in a simulation run allows reducing the number of possible

cache states considerably compared to a static cache analysis covering all possible program paths. Moreover, the effects of a memory access can be estimated even if the compiler did not provide debug information for it. This is the major advantage of the proposed technique, as no relation between source-level variables and memory accesses in the machine code is needed.

The proposed technique is a trade-off between precisely knowing when a memory access happens, but without knowing its exact destination, and potentially missing some accesses, but reconstructing them more precisely. The former concept is the major contribution of this work, while the latter is typical for all previous publications in this area.

### D. Dynamic Abstract Cache Simulation

The final model for source-level simulation consists of the following items:

- The application source code with annotations describing the binary-level basic block matching a source-level statement.
- The path simulation code which simulates the transitions between basic blocks and dynamically selects memory annotations.
- An abstract cache model which uses the concepts of static cache analysis to classify the annotated memory accesses.

During the execution of the annotated software, the memory references obtained from the machine code during model creation are fed into the abstract cache model. Thus, the cache model always maintains the cache state for the current position of the program counter. As this state reflects only the previously executed portions of the program, it is more precise than the abstract state determined by a completely static cache analysis for identical program points.

Based on the dynamically determined abstract cache states, some memory accesses can be classified as cache hit, some can be classified as cache misses and some cannot be classified. The latter is the result of the abstraction used by the dynamic analysis, since memory accesses are described by address ranges instead of single addresses. This means that the result of the analysis is only a range of cache hits and misses, since every unclassified access could be a hit or a miss. While this reduces the accuracy of the determined cache hit and cache miss numbers, it also allows tweaking the simulation for various purposes during architecture exploration. For instance it is possible to simulate the best case and the worst case execution of a program for a fixed set of input values.

Another benefit of the dynamic abstract cache simulation is that it can be easily coupled with models using a different level of abstraction. For instance, the memory accesses generated by an ISS can also be passed to an abstract cache to model a shared cache. Furthermore, the effects of program parts only available in binary form, such as library functions, can also be considered during a source-level simulation. As the path simulation code will model binary-level control flow for these program parts, their effect on the cache contents is simulated as well. Nevertheless, data-dependent control flow might not be simulated accurately. So the effects of the program parts that are only available in binary form might be estimated less precise, but their effect on the cache hits and cache misses generated by the application code can be determined accurately.

## V. Experimental Evaluation

### A. Implementation and Test Setup

The presented approach for source-level cache simulation has been implemented and tested for ARM target processors. For the experimental evaluation, the programs of the Mälardalen WCET benchmark suite [21] were used. The ARM target binaries for the experiments were generated using the Mentor Graphics Sourcery CodeBench Lite Edition compiler toolchain [22], which is based on the **gcc** toolchain. The highest level of compiler optimizations was used (`-O3`) for compiling the benchmarks. The compiler performed many high-level optimizations which significantly modified the structure of the programs, e.g. Function Inlining, Dead Code Elimination and partial Loop Unrolling.

The proposed source-level technique was implemented using existing frameworks for binary-level static analysis and source-level annotation of machine code properties. The path simulation code and the memory annotations for the example programs were generated as described in the previous section. We then compiled the annotated source code and linked it with a library containing a cache model identical to models used in static cache analysis. The resulting host-compiled binary was executed on a standard Linux workstation to determine the number of data cache hits and data cache misses. For all experiments, we used an 8kB 2-way set-associative data cache with a line size of 16 bytes and using the LRU replacement policy. Other cache configurations are supported as well and can be easily evaluated by changing the respective parameters of the cache model.

To validate the results, the ARM target binaries were also executed using QEMU [23] as instruction set simulator. QEMU was used to capture memory traces for the benchmarks which contained all information about the memory accesses executed by the target machine code. With these traces, the cache behavior of the example program was determined using an offline cache model.

### B. Results and Discussion

The results of the experimental evaluation can be found in Table I. Benchmarks for which the compiler generated trivial optimized code, like programs with a single basic block and no memory accesses, or data-dependent loops for which the source code was not available (software floating point division) were not considered in the evaluation. From the remaining programs, the ten benchmarks with the largest memory traces were selected. For each of the selected programs, the total number of instructions executed, the number of data cache hits, the number of data cache misses and the cache miss rate are shown for the ISS-based execution and the source-level simulation. Furthermore, the number of unclassified cache accesses and the error rate for the number of predicted cache misses are included for the source-level simulation. For this estimation, all unclassified accesses were considered to be cache hits.

As can be seen from the results, the proposed technique is able to accurately model the memory accesses of a program on the target architecture. In most cases, the source-level simulation is also able to classify all data cache accesses as hits or misses. Nonetheless, even for a best case or a worst case assumption, meaning counting all unclassified accesses as cache hits or cache misses, the proposed source-level estimate is still reasonably accurate while being more general than existing techniques.

As the cache model was not integrated into the reference ISS, a meaningful comparison of the different approaches with respect to simulation performance was not possible. Therefore the source-level simulation was also compared to the commercial simulator Synopsys CoMET, which is based on binary translation like QEMU. This comparison yielded similar results with respect to the number of cache hits an misses. The

| | ISS | | | | Source-Level Simulation | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Benchmark | Instructions | Hits | Misses | Miss Rate | Instructions | Hits | Misses | Unclassified | Miss Rate | Error |
| recursion | 1852 | 662 | 40 | 0.0570 | 1852 | 662 | 40 | 0 | 0.0570 | 0.00% |
| ns | 1897 | 468 | 158 | 0.2524 | 1898 | 468 | 158 | 0 | 0.2524 | 0.00% |
| select | 3096 | 1267 | 31 | 0.0289 | 3454 | 665 | 26 | 767 | 0.0178 | 16.13% |
| nsichneu | 4505 | 1994 | 13 | 0.0065 | 4537 | 2008 | 11 | 8 | 0.0054 | 15.38% |
| fdct | 4649 | 3153 | 66 | 0.0205 | 4649 | 3153 | 66 | 0 | 0.0205 | 0.00% |
| jfdctint | 6077 | 3601 | 264 | 0.0683 | 6031 | 3601 | 260 | 0 | 0.0673 | 1.52% |
| crc | 12129 | 589 | 543 | 0.4797 | 12129 | 559 | 519 | 54 | 0.4585 | 4.42% |
| ndes | 19926 | 7565 | 238 | 0.0305 | 19926 | 6559 | 182 | 1094 | 0.0232 | 22.22% |
| edn | 30178 | 11874 | 582 | 0.0467 | 30173 | 11867 | 580 | 0 | 0.0466 | 0.34% |
| matmult | 51339 | 15808 | 3011 | 0.1600 | 51339 | 15848 | 2971 | 0 | 0.1579 | 1.33% |

TABLE I: Experimental Results of Data Cache Simulation

performance of the source-level simulation was between one and two orders of magnitude faster for a single execution of the benchmark programs. Repeated execution of the programs using a loop around the main function reduced the simulation speedup, as the overhead created by binary translation can amortize over repeated execution of the same code areas, while the overhead induced by source-level path simulation remains constant.

During the experimental evaluation, only a single-core processor with a single private L1 data cache and executing a single program has been considered. However, the dynamic estimation of cache states allows the consideration of external influences when simulating the cache behavior. Examples for such influences are task switches in a multitasking operating system or the use of shared caches in multiprocessor systems. To model these effects in a source-level simulation, parallel software components must make use of a common cache model. The presented approach can consider these interactions by using a synchronization scheme, like the one presented in [24]. By modeling this interactions on a higher level of abstraction, i.e. not for every processor cycle, source-level simulation has the potential for a significant performance increase when compared to a instruction-based simulation at the machine code level.

## VI. Conclusion

This paper presented a novel approach for the precise host-compiled source-level simulation of data caches. In contrast to previous approaches for source-level simulation, memory access annotations are not directly added to the source code of a program. Instead, annotations are selected dynamically by path simulation code which reconstructs the binary-level control flow. The values for these annotations are determined using a static binary-level interval analysis of processor registers. As the resulting memory annotations do not necessarily point to a unique memory address but to address intervals, a standard functional cache model cannot be used to estimate cache statistics. Therefore the presented technique makes use of an abstract cache model which approximates the actual cache state to determine the cache behavior.

Experimental results have shown that the presented hybrid approach can model the cache behavior accurately. Notably, the presented technique can handle complex compiler optimizations and does not rely on compiler-generated debug information to simulate memory accesses. Working on the source code level allows a seamless integration of the presented technique into established frameworks for system-level performance analysis and design space exploration for embedded systems. Furthermore, a significant boost in simulation performance can be expected due to the native execution of the embedded software on a simulation host.

## Acknowledgments

This work has been partially supported by the BMBF project SANITAS under grant 01M3088C and by the ITEA2 project VERDE under grant 01|S09012A.

## References

[1] "Synopsys CoMET," http://www.synopsys.com/.
[2] "Open Virtual Platforms (OVP)," http://www.ovpworld.org.
[3] "Dinero IV," http://pages.cs.wisc.edu/~markhill/DineroIV/.
[4] H. Cass and P. Sainrat, "OTAWA, a framework for experimenting WCET computations," in *European Congress on Embedded Real-Time Software (ERTS), Toulouse*, 2006.
[5] "AbsInt aiT WCET Analyzer," http://www.absint.com/ait.
[6] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252.
[7] C. Ferdinand and R. Wilhelm, "Efficient and Precise Cache Behavior Prediction for Real-Time Systems," *Real-Time Systems*, vol. 17, pp. 131–181, 1999, 10.1023/A:1008186323068. [Online]. Available: http://dx.doi.org/10.1023/A:1008186323068
[8] T. Meyerowitz, A. Sangiovanni-Vincentelli, M. Sauermann, and D. Langen, "Source-Level Timing Annotation and Simulation for a Heterogeneous Multiprocessor," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2008)*.
[9] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-Performance Timing Simulation of Embedded Software," *45th Design Automation Conference (DAC 2008)*.
[10] K.-L. Lin, C.-K. Lo, and R.-S. Tsay, "Source-Level Timing Annotation for Fast and Accurate TLM Computation Model Generation," *15th Asia and South Pacific Design Automation Conference (ASP-DAC 2010)*.
[11] J. Castillo, H. Posadas, E. Villar, and M. Martinez, "Fast Instruction Cache Modeling for Approximate Timed HW/SW Co-Simulation," in *GLSVLSI '10: Proceedings of the 20th Great lakes symposium on VLSI*.
[12] Z. Wang, K. Lu, and A. Herkersdorf, "An Approach to Improve Accuracy of Source-Level TLMs of Embedded Software," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2011)*.
[13] A. Bouchhima, P. Gerin, and F. Petrot, "Automatic Instrumentation of Embedded Software for High Level Hardware/Software Co-Simulation," *14th Asia and South Pacific Design Automation Conference (ASP-DAC 2009)*.
[14] E. Cheung, H. Hsieh, and F. Balarin, "Fast and Accurate Performance Simulation of Embedded Software for MPSoC," *14th Asia and South Pacific Design Automation Conference (ASP-DAC 2009)*.
[15] Z. Wang and A. Herkersdorf, "An Efficient Approach for System-Level Timing Simulation of Compiler-Optimized Embedded Software," *46th Design Automation Conference (DAC 2009)*.
[16] E. Cheung, H. Hsieh, and F. Balarin, "Memory Subsystem Simulation in Software TLM/T Models," in *14th Asia and South Pacific Design Automation Conference (ASP-DAC 2009)*.
[17] H. Posadas, L. Díaz, and E. Villar, "Fast Data-Cache Modeling for Native Co-Simulation," in *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*.
[18] A. Flexeder, M. Petter, and H. Seidl, "Analysis of Executables for WCET Concerns," Technical Report TUM-I0838, Technische Universität München, 2008.
[19] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Fast and Accurate Source-Level Simulation of Software Timing Considering Complex Code Optimizations," *48th Design Automation Conference (DAC 2011)*.
[20] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Dominator Homomorphism Based Code Matching for Source-Level Simulation of Embedded Software," in *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2011)*.
[21] "Mälardalen WCET research group WCET Benchmark Suite," http://www.mrtc.mdh.se/projects/wcet.
[22] "Mentor Graphics Sourcery CodeBench Lite Edition for ARM," http://www.mentor.com/embedded-software/sourcery-tools.
[23] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the USENIX Annual Technical Conference*, 2005.
[24] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Fast and Accurate Resource Conflict Simulation for Performance Analysis of Multi-Core Systems," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2011)*.