Associative Caches in Formal Software Timing Analysis

Fabian Wolf[*]
Volkswagen AG, Wolfsburg, Germany
fabian.wolf@volkswagen.de

Jan Staschulat
Technical University of Braunschweig, Germany
jans@ida.ing.tu-bs.de

Rolf Ernst
Technical University of Braunschweig, Germany
ernst@ida.ing.tu-bs.de

ABSTRACT

Precise cache analysis is crucial to formally determine program running time. As cache simulation is unsafe with respect to the conservative running time bounds for real-time systems, current cache analysis techniques combine basic block level cache modeling with explicit or implicit program path analysis. We present an approach that extends instruction and data cache modeling from the granularity of basic blocks to program segments thereby increasing the overall running time analysis precision. Data flow analysis and local simulation of program segments are combined to safely predict cache line contents for associative caches in software running time analysis. The experiments show significant improvements in analysis precision over previous approaches on a typical embedded processor.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: [Real-time and embedded systems]; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*; C.4 [**Performance of Systems**]: Performance Attributes

General Terms

Algorithms, Performance, Verification

Keywords

Cache Analysis, Embedded Software, Real-Time, Timing Analysis

1. INTRODUCTION

Embedded software running time is typically not unique but is bound by intervals which result from data dependent behavior, environment timing and target system properties. In system design automation, these intervals are used in many ways. In some cases, only the worst case is of interest, e.g. for single processor schedulability analysis, in another context both best and worst cases are relevant, such as for multiprocessor scheduling analysis or for variable life time analysis as used for memory assignment. In all these cases, running time intervals of the individual software processes are fundamental data needed to analyze system behavior. Profiling and simulation tools such as [9] are the state-of-the-art in industry but since exhaustive simulation is impractical regarding test patterns selection for the conservative running time bounds, simulation results can only cover part of the system behavior.

With growing importance of embedded software, formal analysis of running time intervals has met increasing interest in the EDA and real-time systems communities. Major improvements were the introduction of implicit path enumeration and the inclusion of cache analysis [8]. In [2, 10], data cache analysis has been included using abstract interpretation. While all approaches are conservative, i.e. all possible timing behavior is included in the resulting intervals, the main differences are in the architecture features that are covered by the hardware model and the width of the conservative interval. The closer this interval to the real running time bounds, the higher is the practical use of formal analysis.

Cache hits or misses depend on the mapping of programs and data to memory, and dependencies can span a whole process or even several process executions. This makes simulation particularly risky since the user cannot anticipate the effects of optimization and address mapping in compilers and linkers and, therefore, cannot provide the necessary simulation patterns to test critical cases. On the other hand, cache performance is too significant to be neglected in running time analysis of software on advanced target architectures. Previous research approaches for running time and cache analysis work on basic blocks as elements. These approaches identify the cache lines used by a basic block and compare the different basic block cache tables to identify upper and lower bounds for the number of cache line replacements, hits and misses per basic block execution. Then, based on implicit or explicit program path analysis, bounds on the total cache hits and misses are determined. These approaches may get too expensive because the number of basic blocks can be very high.

In earlier papers [13], we have demonstrated how to extend analysis elements from basic blocks to larger program segments thereby significantly improving both path analysis and architecture modeling precision. The approach is based on local program segment execution, i.e. cycle true simulation. These program segments may contain loops where cache lines are repeatedly replaced by others. Therefore, table comparison alone is not sufficient. In [12] we have presented a preliminary technique which combines local cache tracing with global data flow analysis. In this previous approach we have only determined bounds for the number of cache

[*]This work was performed for the Technical University of Braunschweig, Germany. Since September 2001, Fabian Wolf has been working for Volkswagen AG, Wolfsburg, Germany.

misses for simple direct mapped caches. In this paper, we present significant improvements to this approach. We extend it to set associative caches and consider the impact of the determined cache hits and misses on the running time interval of a process. We consider the majority of effects of compiler optimizations in static software analysis and present detailed experimental results.

The rest of this paper is structured as follows: Previous work is reflected in section 2. Local cache simulation for program segments is explained in section 3 before the methodology for data flow analysis based cache line content prediction is introduced in section 4. The experiments are presented in section 5 before we conclude in section 6.

2. PREVIOUS WORK

Trace based cache simulators have been used in modeling computer architecture for many years. Due to the well-known drawbacks of simulation for real-time guarantees, research approaches focus on static cache analysis. Some of the most important approaches are explained in the following.

In the running time analysis framework in [8], the actual state of the instruction cache depending on the flow of the program is modeled. A simple conflict graph for the cache lines referenced by basic blocks is sufficient for direct mapped caches while associative cache analysis requires a more complex cache state transition (CSTG) graph. The CSTG is used in a so called implicit program path enumeration, an efficient technique that maps timing analysis to a min/max optimization problem which is then solved by integer linear programming (ILP). For higher associativities, the problem becomes prohibitively complex because the problem size grows more that exponentially with the number of basic blocks mapped to a set. The integer linear programming (ILP) problem that computes the running time interval on the process level is already NP-complete. Moreover, implicit enumeration does not exclude many of the false paths of a program and their resulting cache reference sequences, such that this method tends to result in wider bounds than necessary. This approach presented in [8] does not consider data caches.

In [5], a timing analysis tool focusing on the instruction cache analysis part is presented. The problem size is reduced by a clustering of basic blocks in the cache conflict graph mapped to the same cache line. This is an improvement, but cache lines have a very limited size, e.g. 16 instruction words, so the clusters cannot be very large. Data caches and access addresses are not analyzed.

The approach presented in [7] uses data flow analysis for the determination of data dependent access addresses. This can be very useful for the prediction of data cache behavior. The main assumption is that not all data accesses with complex address expressions have to be treated as cache misses. ”Use-/define chains” [1] for access addresses are derived to determine whether address expressions are just depending on constants. In this case, the data access address can be predicted reducing the number of potential cache misses. The framework this approach is embedded in uses an architecture where the instruction and data word size are equal to the size of a cache line, so the improvements through spatial locality have a limited impact.

For many applications, the instruction cache achieves high hit rates for repetitive loops because the loop body is small enough to fit entirely into the cache. Only the first iteration will lead to cache misses while subsequent iterations will lead to hits. In the static cache analysis methodology in [3], a method to analyze the control flow of a program by statically categorizing the cache behavior of each instruction to *always hit*, *always miss*, *first hit* or *first miss* is introduced. The approach looks at the loop bodies starting at the innermost loop nest level. At each level, if the loops fits entirely into the cache, then only the first iteration leads to misses while the subsequent iterations lead to hits. If the loop does not entirely fit into the cache, portions of the loop will always miss the cache due to conflicts. The running time bounds for the loop can be determined. After analyzing the innermost loop, the next loop nest level can be considered. For input data dependent control structures, first miss scenarios have to be assumed for the following instructions. This can have a significant impact on loop nests. Path information relating different loop nests of the program is not considered, so the predicted bounds may be too conservative.

The approach in [2] describes static instruction and data cache analysis using abstract interpretation. It has been refined in [10]. Starting from the memory accesses of the program and their influence on the caches a collecting semantic is built. This collecting semantic assigns all possible cache states to every given memory access. The *program analyzer generator (PAG)* builds the control flow graph from the executable program under investigation. For every node in the control flow graph two analyzes are implemented. The *must*-analysis for the node in the control flow graph determines all memory blocks that must be in the cache for every possible execution of the program. The *may*-analysis for the node delivers all memory blocks that may be in the cache at this point. The memory accesses can be categorized to *always hit (ah)*, *always miss (am)* or *not classified (nc)*. For conservative running time analysis, (nc) are treated as (am) for the upper bound and as (ah) for the lower bound. This methodology is still based on the analysis granularity of single instructions or basic blocks.

3. LOCAL CACHE SIMULATION

In [12], an analysis technique for direct mapped caches has been proposed. It delivers tight bounds on the number of cache hits and misses for a single process on a given target architecture. We review and refine this basic approach, extend it to set associativity and describe the impact of the determined cache hits and misses on the process-level running time intervals. These are crucial extensions to the previously presented technique that enable an application in embedded system design automation.

3.1 Local Running Time Model

Previous approaches like [8] are based on the execution count and the running time of the basic blocks of a program which are then combined in some global path analysis, such as implicit path enumeration. The basic block running time can be determined by adding up the running times for each instruction in a basic block, possibly with upper and lower bounds in case of data dependent instruction running time. This leads to wide intervals in case of architectures with pipelines and caches because it is still assumed that all executions of one instruction take the same time. Precise modeling of individual basic blocks by local simulation only solves part of the problem since cache effects, pipelining and register allocation extend over basic block borders. This results in a dependency of the running time on the program path through a sequence of basic blocks. A startup overhead covering overlapping execution for all possible entry paths into the basic block has to be considered.

To obtain higher precision, the analysis should be extended from basic blocks to longer program segments consisting of sequences of basic blocks whenever possible. On the left side of figure 1, the analysis methodology based on the granularity of basic block running times is shown. Overheads representing conservative assumptions for cache misses, pipeline stalls and register spills have to be assumed for every basic block leading to high overestimations but the control flow between these basic blocks is predictable.

A closer look at the program segment shows that the example has a unique, data independent execution path only. A program segment with such a property can be treated like a generalized basic block. On the right side of figure 1 this program segment is shown. Functional constraint annotation becomes obsolete for this program segment. As we apply local *program segment simulation*, this approach only needs a startup overhead for the segment running time while intermediate basic block transitions can exactly be modeled. In summary, the basic block overhead as a main source of overestimation can be significantly reduced using an identification of suitable longer program segments and the extension of the basic block running time model to program segments.

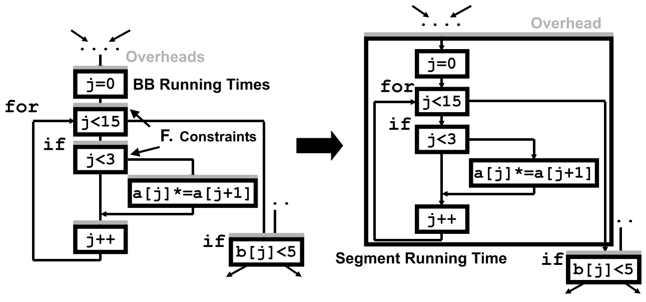


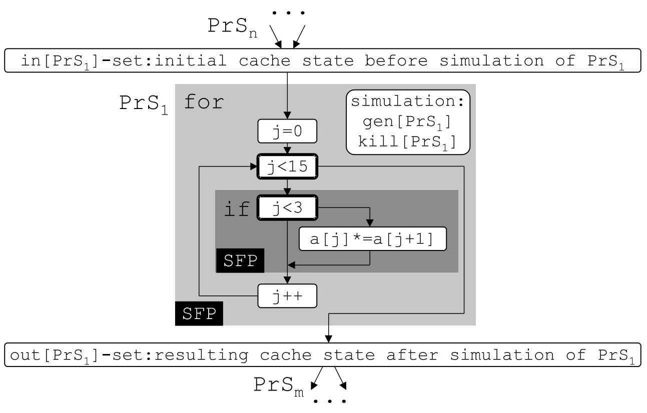
Figure 1: From basic blocks to program segments

3.2 From Basic Blocks to Program Segments

In previous research, we have developed efficient techniques to identify *program segments (PrS)* with unique paths and exploit such segments to reduce analysis problem size and safely reduce timing intervals [11, 13]. Large parts of typical embedded system programs have a single path that is independent of input data [14]. An example is given in the graph in figure 2, other examples may be an FFT or an FIR filter. The control flow of the loop is independent of input data. This path may wrap around many loops, conditional statements and even function calls that are used for source code structuring and compacting. A PrS has a *Single Feasible Path (SFP)*, when paths through the segment are not depending on input data [11, 13].

The key to finding SFP program segments (SFP-PrS) is to distinguish between input data dependent control flow and source code structuring aids. When data dependencies are more difficult to determine than in figure 2, a depth first search algorithm on the syntax graph combined with a symbolic simulation of basic blocks can be applied. In [8], path analysis may be accurate, but it requires much designer interaction even for SFP-PrS while it still does not deliver the precision for program segment running time and cache behavior implied in local simulation. For SFP-PrS, local cache simulation automatically chooses the one correct path and exploits the basic block sequence and memory footprint as well as the resulting cache access sequence without any designer interaction. Local simulation can cover several adjacent or nested SFP-PrS.

Most practical systems also contain non-SFP parts. A program segment has *Multiple Feasible Paths (MFP)*, when paths through its control structure depend on input data. This is the state-of-the-art in basic block based analysis and the according methodologies [8] can be used. An ILP solver computes the process-level running time intervals on the analysis granularity of SFP-PrS instead of basic blocks. The running times for the SFP-PrS nested in the potential paths through the MFP-PrS can be determined using local simulation in a hierarchical path analysis approach [11, 13].



PrS$_n$
in[PrS$_1$]-set:initial cache state before simulation of PrS$_1$
PrS$_1$ for
simulation: gen[PrS$_1$] kill[PrS$_1$]
j=0
j<15
if j<3
a[j]*=a[j+1]
SFP
j++
SFP
out[PrS$_1$]-set:resulting cache state after simulation of PrS$_1$
PrS$_m$

Figure 2: Cache set replacement in SFP simulation

3.3 Set Associativity in Cache Modeling

Instruction and data caches consist of lines that can contain more than one word. The cache line is treated as a whole on every reference. For an associativity of n, n cache lines are summarized to a cache set. The set is referenced by the cache index. Due to the calculation method of the cache index by using the least significant bits of the access address, cache sets are independent. Memory can be divided into line blocks [8], according to the cache lines the parts of the memory segments are mapped to. When more than one line block is mapped to a cache set, n line blocks can be stored before the replacement strategy decides which cache line is discarded.

3.4 Data Caches

Data access addresses are often given as an offset to a fixed base address [7]. When the offset is input data dependent, two misses are assumed, one for the loading of the data of the unknown address and one for the potential replacement of cache line contents that could have been leading to a hit. If the data access address is composed by the base address, constants and variables resulting from a local access sequence in an SFP-PrS. e.g. as for the array $a[\ ]$ in figure 2, this is referenced to as a *Single Data Sequence (SDS)* [14] which is quite common in embedded systems. Data cache behavior of SDS can be determined by local simulation of the program segment.

3.5 Program Segment Cache Evaluation

Simulation for SFP-PrS starts with the best case and worst case assumptions being first hit/miss regarding the contents at the beginning of the PrS. Best case and worst case bounds for an n-way set associative cache are delivered by the established cache simulator DINERO III [6]. It models the associative instruction and data cache behavior of the local address sequence.

In data flow analysis [1], the $gen[bb]$-set contains the generated definitions for a variable in a basic block (bb) while the $kill[bb]$-set contains the destroyed definitions. The $in[bb]$-set is the set of definitions reaching and the $out[bb]$-set the set of definitions leaving the basic block. We apply this methodology to complete cache sets consisting of n lines in the following.

Definition 1 *A cache definition is a content modification of the cache set.* □

A definition for a cache set can result from a miss of the instruction cache, a miss when reading the data cache and every data cache write. On a cache hit on either cache, a definition occurs when the priorities of the cache lines in a set are changed.

Definition 2 *The $gen_{set}[PrS]$-sets are the sets containing the last definitions for each cache set when executing the PrS.* □

This means on local conflict misses, the definition generated by a replaced line block is removed. The $gen_{set}[PrS]$-sets can be computed by local simulation.

Definition 3 *The $kill_{set}[PrS]$-sets are the sets containing the destroyed definitions for the cache sets of the disjoint PrS.* □

A definition from a disjoint PrS is destroyed when the same cache set is defined by the current PrS and the line block from the disjoint PrS is removed from the set as it was chosen by the replacement strategy. The $kill_{set}[PrS]$-sets can be computed by local simulation.

Definition 4 *The $in_{set}[PrS]$-sets are the sets containing the cache set definitions reaching the PrS.* □

The $in_{set}[PrS]$-sets potentially reduce worst case overhead assumptions for the beginning of the local simulation in further analysis.

Definition 5 *The $out_{set}[PrS]$-sets are the sets containing the cache set definitions leaving the PrS.* □

In figure 2, we can see the execution of an SFP program segment containing several basic blocks. The cache contents the simulation starts from corresponds to the $in[PrS]$-set while the cache set replacement during execution defines $gen[PrS]$-set and $kill[PrS]$-set. The $out[PrS]$-set corresponds to the cache state after execution. Running times for the PrS can be computed by local simulation before a process-level solution is computed with an ILP solver. This is described in section 4.4.

4. GLOBAL FLOW ANALYSIS

Even for the first reference of a cache set in the simulation of a program segment (PrS), the assumption of a cache miss is too pessimistic. The line block might already be in the cache set from a previous execution of the current PrS or from a loading of the line block by a PrS mapped to the same line block. This can lead to cache hits even for the first reference in a PrS. This propagation of definitions is referred to as the *Set Definition Propagation* of a program segment $SDP[PrS]$. We utilize well-established work in global flow analysis from compiler design [1] for computation.

4.1 Cache Set Content Prediction

In figure 3, the mapping of the address space of a program to line blocks referring to the cache sets (CS) according to [8] can be seen. Path analysis has found four SFP-PrS where cache behavior can be simulated while the control flow from PrS_1 to PrS_2 and PrS_3 cannot be predicted (MFP-PrS). However, line block 1 will be present in the set at the beginning of PrS_2 because of the previous execution of PrS_1 that loads the complete cache line including parts of the code for PrS_2. Line block 2 may be present because of a previous execution of PrS_2 itself or an execution of PrS_3, when it has not been replaced. This effect is valid for direct mapped caches as well as for higher associativities. In general, overlapping cache sets lead to predictable cache line contents.

The effect reduces the worst case assumption of empty $in_{set}[PrS]$-sets. An analysis technique combining local simulation and static propagation analysis in three steps is proposed. The major differences of this methodology to *must/may* analysis [2, 10] are a conservative path analysis for SFP-PrS and the data flow analysis granularity of program segments instead of single instructions or basic blocks.

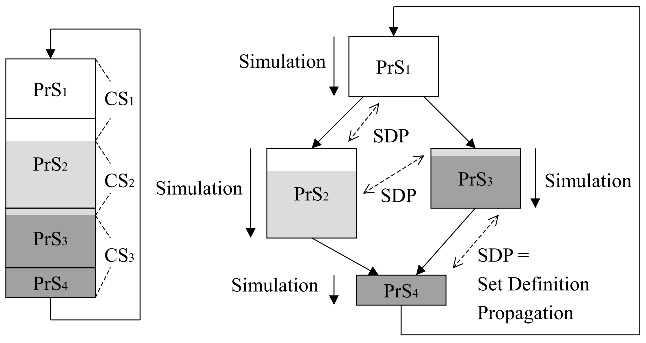


Figure 3: Set Definition Propagation

4.2 Hybrid Prediction Approach

In a **first step**, cache behavior is simulated for every PrS starting with a first miss scenario meaning empty $in_{set}[PrS]$-sets. A first hit scenario is needed for the best case analysis. Local cache hits and misses for the address sequence of the PrS are found by local simulation. The $gen_{set}[PrS]$-sets and $kill_{set}[PrS]$-sets can be computed.

In the **second step**, the data flow analysis defines $out_{set}[PrS]$-sets from $gen_{set}[PrS]$-sets and $kill_{set}[PrS]$-sets given by simulation and $in_{set}[PrS]$-sets.

$$out_{set}[PrS] = gen_{set}[PrS] \cup (in_{set}[PrS] - kill_{set}[PrS])$$

This means the set references leaving the PrS $out_{set}[PrS]$ are composed by the set references within the PrS plus the references entering the PrS which are not replaced. A second set of equations is given by the fact that the $in_{set}[PrS]$-set is defined from the predecessor $out_{set}[PrS]$-sets.

$$in_{set}[PrS] = \bigcap_{P \in pred(PrS)} out_{set}[P]$$

For worst case analysis, only definitions occurring on all previous PrS are propagated, so the intersection operator $\cap$ is used. For best case analysis, the union operator $\cup$ needs to be applied. The equations for every cache set can be solved by iterative forward data flow analysis algorithms [1]. The $in_{set}[PrS]$-sets are assumed to be empty in the first iteration. When data flow analysis is finished, the $in_{set}[PrS]$-sets either contain a subset of the propagated definitions for worst case analysis or a superset of the propagated definitions for best case analysis.

In the **third step**, worst case analysis finds unique definitions in the $in_{set}[PrS]$-sets that PrS simulation has classified as misses for the first reference due to the conservative cache state, namely empty $in[PrS]$-sets at the beginning of the PrS. Similarly, best case analysis finds the maximum number of definitions reaching the PrS. The initial first hit assumptions not being in the $in[PrS]$-sets can be replaced by misses improving the best case analysis bound.

4.3 Evaluation of Flow Analysis Results

A definition in the $in_{set}[PrS]$-set represents an instruction or data word that is present in the set. When the cache line referenced by the definition in the $in_{set}[PrS]$-set is classified as a first miss by simulation, the assumption in the first step is too conservative.

When the definition in the $in_{set}[PrS]$-set is from another program segment (PrS) whose execution would load the current line block — an execution of PrS_3 loads CS_3 for PrS_1 in figure 3 — the first miss (compulsory) for this line and PrS can be removed because the memory block is loaded in the other PrS and not killed.

When the unique definition in the $in_{set}[PrS]$-set is from the memory block of the program segment (PrS) itself, all misses (compulsory) for this line and PrS can be removed, except for the miss of the first execution. For associative caches, just removing the first miss is not sufficient because the referenced set contains more than one line. The global analysis only propagates definitions for a set, not for a cache line, so the replacement strategy can be considered in the local simulation of the program segments. Per predicted cache hit, the running time of one cache miss can be subtracted from the conservative program segment startup with empty $in_{set}[PrS]$-sets. This is done by means of replacing the conservative compulsory misses with the newly predicted hits in a new local simulation where pipeline and register set are still modeled conservatively assuming a cold start for the beginning of the program segment. When this approach is used, global cache effects can be integrated into the overheads of the local program segment running time intervals determined by simulation. For this purpose, a slight modification of the cache simulation tool is necessary. DINERO III has been extended with the possibilities to preset, dump or directly modify the cache behavior of a given address sequence. It can model least recently used and round robin replacement.

4.4 Running Time Calculation

For the MFP-PrS and the process-level result, the approach from [8] can be applied on the simulated lower level SFP-PrS running times including several basic blocks. Implicit path enumeration can bound the longest path through the MFP-PrS. The startup overhead has been reduced using the flow analysis results. In further work, more analysis techniques from the real-time research community can be added to further improve the precision of the SFP program segment startup overhead with respect to pipeline and register set behavior. The complex cache analysis using a transition graph is not needed because the cache behavior is modeled using the startup overheads.

4.5 Compiler Optimization and Code Motion

As the presented analysis approach is based on the source code level, local and global compiler optimization can have significant effects on the running time intervals. It has been shown that most compiler optimizations are confined to the level of basic blocks [4], so they do not influence the results when the same optimizations are used for analysis. Code motion, e.g. removing loop invariant code from a loop body and inserting it before or after the loop introduces more problems. For the analysis granularity of basic blocks, these optimizations have to be avoided to deliver correct results. This leads to wider running time intervals to stay conservative. For the analysis granularity of program segments, code motion from a loop body that stays in the same program segment is allowed. Instruction implementation via assembly library functions can be treated as data dependent instruction running time. Data dependent instruction running time must be treated by formal subtraction or addition of the best case or worst case timing as described in [14]. Running time intervals for the library functions need to be provided by their vendors. Instructions implemented as exceptions and interrupt effects are not included in the program segment running time but must be treated on the process level.

5. EXPERIMENTS

Running time analysis based on the granularity of basic blocks [8] is compared to running time analysis based on SFP-PrS identification and local simulation of PrS. In this experiment, the original intention is to free the designer from functional constraint annotation. For the basic block based approach, this would lead to infinite running time intervals. For this reason, the loop bounds have been annotated for the basic block based approach which already implies designer knowledge about the program under investigation. This is not necessary in the approach based on program segments when loop bounds are input data independent (SFP). For the local simulation of the SFP-PrS as well as the basic blocks BB, a StrongARM (SA-110) simulator with 160 MHz core frequency, 40 MHz bus frequency and 85 ns memory cycle time has been used. The parameters in local cache simulation are selected according to the real parameters of the StrongARM processor which is an associativity of 32, a block size of 32 bytes, 16 kbytes instruction cache and 16 kbytes data cache. This leads to a cache miss penalty of 32 bus cycles. In [8], this penalty for the i960 has been much lower, so overestimations would seem lower for the i960 instead of the StrongARM. The round robin replacement strategy is modeled. No cache state transition graph (CSTG) for the basic block based approach has been used. A CSTG for an associativity of 32 would have been much too complex. Cache analysis has only been done by means of local simulation which leads to high overestimations in a basic block based approach. Cache analysis supported by set definition propagation has only been applied on the granularity of program segments. Both instruction and data cache behavior have been considered.

Four configurations exploring the effects of program segment source code translation from isolated files or connected program segments in one single file as well as the effects of compiler optimizations (GNU -O1) are presented. In the first column of each table, the benchmark program is provided. The exact running time intervals given by an application of exact worst case and best case input data annotated by the designer are shown in the second column. This would not be possible for more complex programs. In the presented cases, these have been determined manually only to serve as a reference for the evaluation of the static analysis approaches. These running time intervals are different for the same input data when compiler optimizations are applied. The third column shows the running time intervals determined on the analysis granularity of basic blocks (ILP on BB) while the next column shows the number of loop bounds (lb) being the functional constraints that have been annotated to avoid infinite running time intervals. In the next column, the running time intervals for the approach using SFP identification (ILP on SFP) are given. In the last column, the running time intervals for the extension of the SFP identification approach with cache analysis using set definition propagation (ILP on SDP) are shown.

In the first configuration, the running time intervals have been determined using isolated local simulation of program segments, i.e. in different source code files after path analysis. No compiler optimizations have been applied. The results are given in table 1.

Table 1: Intervals for isolated PrS, no optimization

Benchmark [μs]	Exact	ILP on BB	lb	ILP on SFP	ILP on SDP
arrcalc	[19.45,20.37]	[4.805,206.9]	4	[3.339,29.92]	[9.20,28.93]
chkdata	[15.62,20.72]	[1.082,226.0]	2	[1.233,152.2]	[9.039,39.82]
bsort	[58.69,104.6]	[13.48,3046]	2	[8.696,1316]	[15.09,846.2]
circle	[47.96,151.1]	[4.269,622.1]	1	[4.287,154.4]	[5.962,153.5]
FIRFilter	[72.15,100.0]	[38.53,2566]	4	[42.99,158.9]	[60.17,136.5]
countsort	[38.10,41.47]	[15.77,1079]	2	[16.28,475.9]	[29.50,290.5]
exchsort	[43.18,43.96]	[17.46,1164]	2	[19.40,237.9]	[30.51,49.34]

In the second configuration, the running time intervals have been determined using local simulation of all PrS and cache analysis in one single file. Worst case startup overheads for the program segments or basic blocks according to the applied analysis granularity have been inserted afterwards. No compiler optimizations have been applied. The results are given in table 2.

Table 2: Intervals for connected PrS, no optimization

Benchmark [μs]	Exact	ILP on BB	lb	ILP on SFP	ILP on SDP
arrcalc	[19.45,20.37]	[2.54,195.7]	4	[3.189,27.30]	[9.186,23.43]
chkdata	[15.62,20.72]	[0.69,172.1]	2	[0.864,78.74]	[8.089,31.77]
bsort	[58.69,104.6]	[12.61,2802]	2	[8.831,1076]	[14.80,459.8]
circle	[47.96,151.1]	[4.26,622.1]	1	[4.287,154.4]	[5.962,153.5]
FIRFilter	[72.15,100.0]	[38.44,2552]	4	[42.99,150.5]	[60.21,122.8]
countsort	[38.10,41.47]	[10.49,788.9]	2	[11.16,295.2]	[20.04,126.2]
exchsort	[43.18,43.96]	[14.62,1116]	2	[19.75,153.2]	[31.67,46.49]

In the third configuration in table 3, the running time intervals have been determined using isolated local simulation of program segments in different input files again. Compiler optimizations with -O1 have been applied.

Table 3: Intervals for isolated PrS, with optimization -O1

Benchmark [μs]	Exact	ILP on BB	lb	ILP on SFP	ILP on SDP
arrcalc	[11.95,12.97]	[1.188,124.7]	4	[1.262,29.71]	[3.881,14.03]
chkdata	[8.792,10.58]	[1.199,167.0]	2	[1.225,116.1]	[2.826,98.72]
bsort	[23.58,27.52]	[4.714,1017]	2	[1.572,693.3]	[13.22,682.7]
circle	[39.27,100.9]	[3.292,480.6]	1	[3.295,123.6]	[5.901,101.4]
FIRFilter	[30.97,37.59]	[13.52,1874]	4	[20.87,353.1]	[29.68,210.0]
countsort	[15.31,15.50]	[6.554,746.5]	2	[7.003,369.8]	[14.48,188.3]
exchsort	[17.20,17.20]	[5.932,622.2]	2	[6.322,250.9]	[9.863,250.6]

In the final configuration presented in table 4, the running time intervals have been determined using local simulation of all PrS and cache analysis in one single source code file. Worst case startup overheads have been inserted afterwards. Compiler optimizations with -O1 have been applied.

Table 4: Intervals for connected PrS, with optimization -O1

Benchmark [μs]	Exact	ILP on BB	lb	ILP on SFP	ILP on SDP
arrcalc	[11.95,12.97]	[1.324,122.7]	4	[1.412,29.40]	[3.881,14.03]
chkdata	[8.792,10.58]	[0.306,94.14]	2	[0.312,48.14]	[3.834,23.61]
bsort	[23.58,27.52]	[5.070,703.9]	2	[2.894,129.2]	[14.69,35.61]
circle	[39.27,100.9]	[3.292,480.6]	1	[3.295,123.6]	[5.901,101.4]
FIRFilter	[30.97,37.59]	[11.28,988.8]	4	[14.35,296.0]	[26.78,198.1]
countsort	[15.31,15.50]	[2.045,208.5]	2	[3.409,75.22]	[8.949,20.75]
exchsort	[17.20,17.20]	[6.067,322.1]	2	[5.644,26.66]	[7.903,24.23]

The experiment reveals that the approach using SFP identification delivers significantly tighter running time intervals than the approach on basic block level. This is an expected result because local simulation for SFP-PrS can consider overlapping basic block effects instead of assuming worst cases for caches at basic block beginnings. These have a significant impact when the StrongARM processor with its high miss penalty is assumed as the target architecture. When cache analysis is applied in combination with SFP identification, the intervals get tighter because many startup overheads for SFP-PrS, especially in nested loops, can be reduced.

Due to the high cache miss penalty compared to the running time for a hit in these configurations, nested loops with embedded MFP can still cause high overestimations because empty caches for the isolated segments in the MFP have to be assumed for every iteration. When SDP cannot predict all of these cache contents, the interval stays relatively wide, e.g. for the *bsort* algorithm. Nested loops with an embedded MFP require the assumption of several cache misses for every iteration. ILP analysis of connected PrS with a later addition of overheads according to the selected granularity delivers tighter intervals than for isolated PrS. More information on cache states is available. The consideration of compiler optimizations leads to more realistic running time intervals with respect to the final solution on the embedded target architecture.

6. CONCLUSION

We have shown how to combine local cache tracing with global data flow analysis in formal cache analysis. The technique matches a processor modeling and path analysis technique which extends architecture modeling from basic blocks to larger program segments thereby increasing the overall analysis precision. The approach has been extended from direct mapped caches to set associative caches and the impact of compiler optimizations has been considered. We have demonstrated that our approach considering both instruction and data cache behavior can significantly increase the analysis precision of state-of-the-art approaches.

7. REFERENCES

[1] A. V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, GB, 1988.

[2] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Journal of Real-Time Systems, Special Issue on Timing Analysis and Validation for Real-Time Systems*, November 1999.

[3] C. Healy, R. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, pages 53–70, January 1999.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publ., 1996.

[5] A. Hergenhan and W. Rosenstiel. Static timing analysis of embedded software on advanced processor architectures. In *Proceedings of Design, Automation and Test in Europe*, pages 552–559, Paris, March 2000.

[6] M. Hill. DINERO III Cache Simulator. www.ece.cmu.edu/ ece548/tools/dinero/src/, 1998.

[7] S.-K. Kim, S. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Proceedings of IEEE Real-Time Technology and Applications Symposium*, 1996.

[8] Y. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[9] Mentor Graphics. *Seamless Co-Verification Environment*. http://www.mentorg.com/seamless/.

[10] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Seperate Cache and Path Analyses. *Journal of Real-Time Systems*, 18(2/3):157–179, May 2000.

[11] F. Wolf and R. Ernst. *Behavioral Intervals in Embedded Software*. Technical University of Braunschweig, Germany, To be published by Kluwer Academic Publishers in 2002.

[12] F. Wolf and R. Ernst. Data flow based cache prediction using local simulation. In *Proceedings of the IEEE High Level Design Validation and Test Workshop*, pages 155–160, Berkeley, USA, November 2000.

[13] F. Wolf and R. Ernst. Intervals in software execution cost analysis. In *Proceedings of the IEEE/ACM International Symposium on System Synthesis*, pages 130–135, Madrid, Spain, September 2000.

[14] W. Ye and R. Ernst. Embedded program timing analysis based on path clustering and architecture classification. In *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD '97)*, pages 598–604, San Jose, USA, 1997.