Data Flow Based Cache Prediction Using Local Simulation

Fabian Wolf, Rolf Ernst
Institut für Datenverarbeitungsanlagen, Technische Universität Braunschweig
Hans-Sommer-Str. 66, D-38106 Braunschweig, Germany
Tel: +49 531 391 3728, Fax: +49 531 391 4587
Email: $\{wolf|ernst\}$@ida.ing.tu-bs.de

Abstract

Accurate cache modeling and analysis are crucial to formally determine program execution time. Current cache analysis techniques combine basic block level cache modeling with explicit or implicit program path analysis. We show how to extend program and data cache modeling from basic blocks to program segments thereby increasing the overall execution time analysis precision. The approach combines architecture simulation, data flow analysis and implicit path enumeration.

1. Introduction

Accurate software execution time analysis is key to high level design validation. Imprecise estimation of execution time and cache behavior increases design risk or leads to inefficient and expensive designs. Profiling and simulation are state-of-the-art in industry but since exhaustive simulation is impractical, simulation results can only cover part of the system behavior. Static analysis is a more complicated but attractive alternative. It provides lower and upper bounds reflecting data dependent control flow as well as data dependent statement execution time. These bounds were wide due to a lack of efficient control flow analysis and architecture modeling techniques. Significant progress in both areas has made formal analysis practical.

One of the problems in formal execution time analysis is cache behavior. 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 or tracing particularly risky since the user cannot anticipate the effects of address mapping in compilers and linkers and, therefore, cannot provide the necessary simulation pattern to test critical cases. On the other hand, cache performance is too significant to be neglected in timing analysis of software running on sophisticated target architectures.

The literature proposes several approaches to cache analysis which will be explained in the following section. They all work on basic blocks as elements. The 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, upper and lower bounds on the total cache hits and misses are determined.

In earlier papers [15, 14], 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. Instead, we present a technique which combines local cache access tracing with global data flow analysis.

Previous work on static cache analysis is explained in section 2. An approach using local cache simulation is presented in section 3 before global cache analysis for a process is explained in section 4. Experiments are presented in section 5 before we conclude in section 6.

2. Previous Work on Cache Analysis

Trace based cache simulators have been used in modeling computer architecture for many years. Research into formal analysis focuses on static cache analysis.

Always Hit/Miss Very early work classifying memory accesses to *always hit* or *always miss* regarding the cache is presented in [1]. A more realistic approach is to assume a miss for the first reference to an address and a hit for every further access because the cache line content may not be present at startup, but after loading on the first miss. This may be nearer to the correct solution, but due to the mixed assumptions, it is not possible to guarantee conservative upper or lower bounds for cache performance.

Cache and Path Analysis Further work on the investigation of the always/first hit/miss scenario has been done in [5]. A method to analyze the control flow of a program by statically categorizing the cache behavior of each instruction is introduced. The approach is tightly coupled to architecture specific pipeline analysis for SPARC, so it is difficult to use it for more general target architectures.

Cache State Transition Graph In the timing analysis framework CINDERELLA by Li and Malik, the actual state of the cache depending on the flow of the program is modeled. In [10], a conflict graph for the cache lines referenced by basic blocks is proposed for direct mapped caches while in [11] a cache state transition graph is built for caches with higher associativity. Implicit enumerations of the transitions in the cache graph and for the program path are used. This leads to a correct solution where all cache states and conflicts within sets are modeled in the cache state transition graph, but the problem size gets very complex. Grouping of basic blocks mapped to the same cache lines [7] can improve the approach regarding accuracy and analysis time, but this still does not contain the sequence of cache references which has a great impact on accuracy and complexity in cache performance prediction. Structural constraints and functional constraints often do not exclude paths and the resulting cache references, so both methods have limited impact and estimated bounds are wide.

Abstract Interpretation The approach in [3] describes static cache analysis by abstract interpretation. A program analyzer generator is developed using abstract cache states to reduce the complexity implied in the approach from [11]. The analysis assumes that some memory blocks *must* be in the cache leading to a cache hit, some memory blocks *may* be in the cache while others are definitely not in the cache.

Use-/Define Chains for Data Access Addresses Data flow analysis can be used for the determination of data dependent addresses [9]. The main assumption is that not all data accesses with complex expressions for the address have to be treated as cache misses. "Use-/define chains" for access addresses to determine whether these expressions are just depending on constants and constant register contents can be applied. 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 equal the size of a cache line, so the improvements through spatial locality cannot be exploited while the determination of access addresses can be used by any tool suite.

3. Trace Based Cache Simulation

A method that delivers tight bounds for the cache behavior of a single process on a given target architecture is proposed. Input data independent program segments covering several basic blocks are found [15, 14] and can be simulated with any trace based cache simulator.

3.1. From Basic Blocks to Program Segments

For trace based cache analysis, executable program segments have to be isolated. The program path analysis model in [11] is established as a standard model for static approaches, while it limits the application of trace based cache simulators to basic blocks using cache tables to determine the number of cache line replacements.

For higher accuracy, *basic block sequences* in program segments (PrS) must be considered. Program properties can be exploited to extend basic blocks to program segments [15, 14]. Large parts of typical embedded system programs have a single path that is independent of input data. An example is given in figure 1. This path may wrap around many loops, conditional statements and even function calls which are used for source code structuring and compacting. A program segment has a *Single Feasible Path (SFP)*, when paths through this segment are not depending on input data.

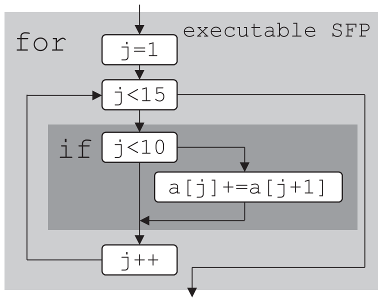

Figure 1. SFP program segment execution

The key to finding SFP segments is to distinguish between input data dependent control flow and source code structuring aids. Input data dependencies can be determined by symbolic simulation of basic blocks. In the approach in [11], path analysis may be accurate, but it requires much designer interaction for SFP program segments while it still does not deliver the program segment latency time and cache behavior. For SFP, execution of this segment with a cache simulator automatically chooses the one correct path and exploits the basic block sequence and cache footprint without any designer interaction.

Most practical systems also contain non-SFP parts. A program segment has *Multiple Feasible Paths (MFP)*, when paths through the program segment depend on input data. Programs can also have context dependent behavior, referred to as process modes given by a subset of input data [14]. For a given context, control structures depending on the subset of input data defined by the context have a *Context Dependent Path (CDP)* and can be treated like an SFP in further analysis, assuming the context dependent input data to be constant for very context.

3.2. The Cache Model

A Harvard architecture for memory and caches is assumed while the concepts are also valid for unified caches. Direct mapped caches are assumed for static analysis in a first approach. Instruction and data cache consist of cache lines that contain several words. A line is completely replaced on a cache miss. This means that the prediction of a line content has a great effect. Due to the calculation method of the cache index [6] by using least significant bits of the access address, cache lines are independent. A *first* or a *last* cache line cannot be identified because mapping restarts from the beginning of the cache after the access address has been incremented beyond the line with the highest index. Memory can be divided into line blocks [11], according to the cache lines the parts of the memory segments are mapped to. Replacement strategies are not needed for direct mapping. We assume a write back data cache while write buffers are currently not considered. Other cache properties only affect access times while this approach focuses on the prediction of cache line contents.

3.3. Access Addresses and Data Caches

The address sequence for extracted program segments can be generated using host tracing and the address mapping after compilation for the target architecture. For program segment beginnings, text and data segment addresses can be read from the map file of the linker. This means instruction access addresses can always be determined while data access addresses may be given as an offset to the base address that can be input data dependent. Two misses are assumed in this case [9], 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 constants and variables resulting from a fixed local access sequence, we call this a *Single Data Sequence (SDS)* [15] if it is part of an SFP. This again extends basic block based approaches.

3.4. Cache Simulation

Simulation starts with the best case and worst case assumptions being first hit/miss for the line regarding the contents at the beginning of the PrS. Best case and worst case bounds for cache hits and misses using DINERO III [8] are delivered. The results for the PrS can be integrated using the methodology from [11] as explained in [14]. Super-scalar architectures as well as branch prediction and code motion may introduce problems for the determination of address sequences. This is left out of scope in our approach.

3.5. Program Segment Cache Evaluation

In classic data flow analysis [2], the *gen*[]-set contains the generated definitions for a variable in a basic block while the *kill*[]-set contains the destroyed definitions. The *in*[]-set is the set of definitions reaching and the *out*[]-set the set of definitions leaving the basic block. In our case, basic blocks have been extended to program segments. We apply this terminology to cache line states instead of variables:

1. A *definition* of a cache line by a PrS is a *miss* of the instruction cache, a *miss* when reading the data cache and every writing to the data cache.
2. The $gen_{line}[PrS]$-set contains the *generated* definitions for the cache lines of the PrS. It can be computed from the local simulation results.
3. The $kill_{line}[PrS]$-set for the current PrS contains the *destroyed* definitions for the lines of the disjoint PrS. A definition from a disjoint PrS is destroyed, when the same cache line is defined and replaced with the current PrS memory contents. This can also be computed from all the local PrS simulation results.
4. The $in_{line}[PrS]$-set is the definitions reaching the PrS.
5. The $out_{line}[PrS]$-set is the definitions leaving the PrS.

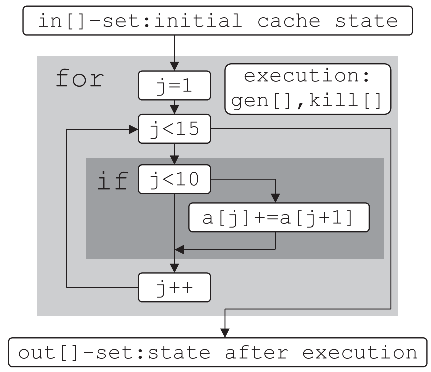

Figure 2. Line replacement in SFP execution

In figure 2, we can see the execution of an SFP program segment. The cache contents the simulation starts from corresponds to the *in*[]-set while the cache line replacement during execution defines *gen*[]-set and *kill*[]-set. The *out*[]-set corresponds to the cache state after execution.

4. Global Flow Analysis

Even for the first reference of a cache line in a PrS, the assumption of a cache miss is too pessimistic. The memory block might already be in the cache from a previous execution of the current segment or from a loading of the line by a different program segment mapped to the same memory block which has not been replaced. This can lead to cache hits even for the first reference in a PrS. This propagation of definitions is referred to as the **Line Definition Propagation** of a program segment $LDP[PrS]$. From here on, we utilize well-established work in global flow analysis as known from compiler design [2, 12] for computation.

4.1. Cache Line Content Prediction

In figure 3, the mapping of the program to three cache lines CL of the instruction cache according to [11] can be seen. Path analysis has found four PrS where cache behavior can be simulated while the control flow from $PrS_{1,3}$ to $PrS_{4,5}$ and PrS_6 cannot be predicted leading to an MFP classification. Cache line 1 will be present at the beginning of $PrS_{4,5}$ because of the execution of $PrS_{1,3}$. Cache line 2 may be present because of a previous execution of the PrS itself or an execution of PrS_6, when it has not been replaced.

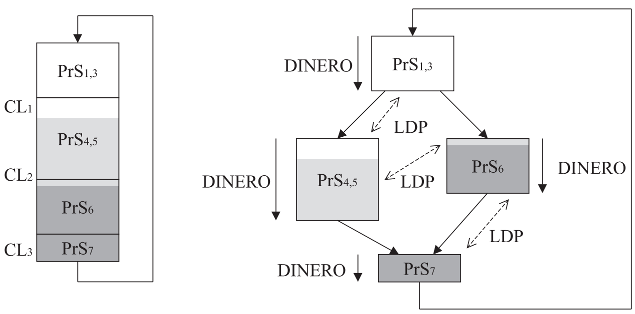

Figure 3. Local simulation and global LDP

This reduces the worst case assumption of an empty $in_{line}[PrS]$-set. A hybrid cache analysis combining simulation and static analysis in three steps is proposed. In the following, we focus on cache misses as the worst case bound. The best case bound assuming cache hits can easily be determined using the same methodology.

4.2. Hybrid Prediction Approach

In a **first step**, cache behavior is simulated for every PrS starting with a first miss scenario, meaning empty $in_{line}[PrS]$-sets. Local cache hits and misses for the address sequence of the PrS are found according to section 3. The $gen_{line}[PrS]$-set and $kill_{line}[PrS]$-set can be computed from the simulation results.

In the **second step**, data flow equations can define $out_{line}[PrS]$-sets from $gen_{line}[PrS]$-sets and $kill_{line}[PrS]$-sets given by PrS simulation and $in_{line}[PrS]$-sets known at PrS beginnings.

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

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

$$in_{line}[PrS] = \bigcap_{P \varepsilon pred(PrS)} out_{line}[P]$$

Only definitions that occur on all previous PrS are propagated, so the intersection operator $\cap$ is used instead of the confluence operator $\cup$. A more formal derivation of this operation can be found in [4]. These equations for the $in[PrS]$-sets and $out[PrS]$-sets can be solved by modified iterative forward data flow analysis algorithms [2]. When the last iteration of this data flow analysis is finished, the $in_{line}[PrS]$-sets contain the definitions resulting from previous misses.

The **third step** can find unique definitions in the $in_{line}[PrS]$-sets that PrS simulation has classified as cache misses for the first reference due to the conservative cache state, namely empty $in[]$-sets at the beginning of the PrS.

- When the definition in the $in_{line}[PrS]$-set is from another PrS whose execution would load the current line — an execution of PrS_6 loads PrS_7 — the first miss for this line and PrS can be removed because the memory block is loaded in the other PrS and is not replaced.
- When the unique definition in the $in_{line}[PrS]$-set is from the memory block of the PrS itself, the first miss for this line and PrS can be removed but one miss for the first execution of the PrS has to be kept.

When information about the cache state are needed at a given point, data flow analysis computes the state at the beginning of the PrS where DINERO starts to compute the local effects according to figure 2. This equals to the computation of a "use-/define chain" with the basic blocks extended to PrS. The previous description is given for the worst case for cache behavior. Best cases can be derived with the same data flow equations analyzed for guaranteed misses instead of hits. Complexity can be a problem in data flow analysis, but due to the path recognition that gives an exact sequence for local definitions most cache state transitions are computed on a local level and do not add to global complexity. Compact data structures and algorithms from well established work [2, 12] are used.

4.3. Global Execution Time Computation

Cache hits and misses lead to a modified execution time t_i of the PrS consisting of the execution time for cache hits and misses $t_i = t_{i,hit} + t_{i,miss}$. The model in [11] is adapted to use PrS instead of basic blocks. Deviating from [11], our approach delivers the overall interval I with execution time t_i and execution count interval $[x_{i,min}, x_{i,max}]$ of the PrS.

$$I = \sum_{i}^{N} \sum_{j}^{n_i} (t_{ij,hit}[x_{ij,min}, x_{ij,max}]_{hit} + t_{ij,miss}[x_{ij,min}, x_{ij,max}]_{miss})$$

The execution count interval $[x_{i,min}, x_{i,max}]$ is determined by implicit path enumeration and ILP solving after our hit/miss scenario has been inserted. The $t_{i,miss}$ are refined to

$$t_{i,miss} = t_i[PrS] + t_{i,misspen} + t_{i,load} - t_{i,loadoverlap}$$

meaning that the execution time t_i is composed by the time for executing the PrS assuming a hit $t_i[PrS]$ plus the penalty for the cache miss $t_{i,misspen}$ plus the time for the cache load $t_{i,load}$ while the time $t_{i,loadoverlap}$ where the cache load is done in parallel can be subtracted. Misses lead to a modified execution time where the best case is that a miss does not block execution, meaning $t_{i,misspen} + t_{i,load} = t_{i,loadoverlap}$. The worst case is that every miss leads to a complete blocking of the processor core, meaning $t_{i,loadoverlap} = 0$.

5. Experiments

5.1. Case Study: Bubble Sort

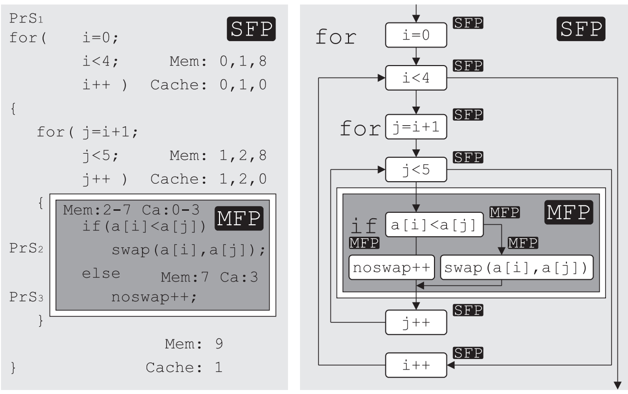

Figure 4. Bubble sort control flow graph

A bubble sort algorithm from [14] and its flow graph with its cache mapping are shown in figure 4. The body of the `for`-loops is classified as MFP due to the unknown input data while the nested loops are classified as SFP. As CDP are treated as SFP, the example of SFP covers all basic block extension effects.

Cache Parameters In this case study, the 128 byte instruction cache uses direct mapping, consists of four blocks 0-3 and four bytes per instruction, so the block size is 32 byte, meaning 8 instructions per blocks. This simple configuration allows us to explain the clustering effect and the line definition propagation. A realistic cache size would imply too complex data structures to show all the previously explained effects using our example.

SFP clustering and DINERO In the example, PrS_1 is executed under any input condition while after the splitting of the path to PrS_2 or PrS_3, the control flow cannot be predicted. The program is compiled and linked, so base addresses of the PrS are available. The isolated PrS are loaded to their original base addresses. DINERO III determines all local cache effects for the PrS. PrS_1 is mapped to the memory blocks 0-2 and 8-9 while the if/else statement is mapped to 2-7. These are all mapped to the cache lines 0-3 which results in conflicts between the PrS. We start simulation for PrS_1 and PrS_2 from empty caches and assume a cache flush whenever we reach a different PrS. As PrS are maximized [14], this overhead is kept as small as possible.

At program startup, first misses load block 0,1 and 2 that are not replaced until PrS_2 and PrS_3 are reached. A cache flush is assumed, so first misses occur for memory blocks 2-7 when executing the `swap()`-code in PrS_2 or for memory blocks 2 and 7 when executing PrS_3. Local conflicts in PrS_2 between block 2 and 6 as well as block 3 and 7 resulting in misses occur for cache lines 2 and 3. Cache lines 0 and 1 are only loaded once and are not replaced. After control flow merges behind the if/else statement, caches are flushed again. First misses occur for loading memory blocks 8, 9 and 0 while the following 0 and 1 conflict with 8 and 9 mapped to the same cache lines before we reach PrS_2 and PrS_3 again. For every following loop iteration, flushes occur at the beginnings of PrS, in our case before PrS_2 and PrS_3 and when they merge again. Hits occur for all other memory accesses that we do not explicitely mention.

Data flow equations For the PrS, the $gen[]$ and $kill[]$-sets are generated from the results of step 1. DFA algorithms [2] are used to compute the $in[]$ and $out[]$-sets.

Propagated Definitions As the memory blocks for the cache lines 0 and 1 in the $in[]$-set of PrS_2 and PrS_3 are disjoint because they result from an execution of PrS_1, no first miss assumptions can be removed for them. For cache line 2, the content results from the execution of the same memory block both PrS_1 and PrS_2 are mapped to, so the first miss assumption can be corrected. Cache line 3 is not replaced and stays in the cache, so it is in the $in[]$-set of PrS_2 and PrS_3. For PrS_2, this has no effect because block 3 is mapped to cache line 3 resulting in a miss while a hit from a previous execution occurs when PrS_3 is executed.

When control flow merges again, the definitions for cache line 0-2 do not leave the if/else statement because they are only referenced in the if-branch [2]. The code defining cache line 3 leaves the if/else, but it only leads to an additional hit for PrS_3 while one miss has to be assumed for its first loading. There is no additional hit in PrS_1 because the memory blocks do not overlap.

5.2. Path Analysis and Local Simulation

In this experiment, path analysis and local simulation of instruction and data cache behavior are investigated. A modified StrongARM simulator has been used for the generation of address sequences [13]. We apply local cache simulation to basic blocks and program segments using DINERO III [8]. The upper bounds are calculated without annotating functional constraints while automatic path recognition was used. For this reason, the accuracy of the results cannot be directly compared to [11]. The presented benchmarks are a packet receiver with cyclic redundancy check, the bubble sort algorithm with a larger data array than in the previous example consisting of 15 elements and a two dimensional image filtering algorithm (smooth). Cache sizes of 32 byte, 1 kbyte and 16 kbyte assuming 32 bit accesses and block sizes of 8 elements are explored. Cache simulation starts from first miss scenarios.

MFP/PrS	32 byte	1 kbyte	16 kbyte
Receiver	8208/4243	-/1046	-/684
Bubble Sort	9858/7733	8858/115	8858/115
Smooth	2833k/2213k	-/70k	-/12k

Table 1. Local Simulation for Cache Analysis

In table 1, the number of cache misses for local simulation of basic blocks (MFP) without automatic path analysis is given on the left side of each table entry <u>MFP</u>/PrS while the results with SFP and CDP identification (PrS) are given on the right side of each table entry MFP/<u>PrS</u>. Where no results are given, analysis of single basic blocks was too complex. The number of cache misses decreases with cache size, especially when path analysis is used because the number of worst case assumptions at basic block beginnings can be reduced as we have to assume worst cases at PrS beginnings. For the bubble sort algorithm, we can see that it is completely placed in the cache if the cache size is 1 kbyte or greater because the number of misses does not decrease with a larger cache. The main result is that larger program segments are very important for cache analysis when using local simulation regarding accuracy and complexity. We conclude that path analysis is key to static cache analysis.

6. Conclusion

We have shown how to combine local cache tracing with global data flow analysis in formal program and data cache analysis. The technique matches a processor modeling and path analysis technique which extends architecture modeling from basic blocks to larger program segment thereby increasing analysis precision. The experiments show improved cache analysis precision. The technique is currently restricted to direct mapped caches but can be extended to higher cache associativity in further work.

References

[1] A. Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming*. Kluwer Academic Publishers, 1989.

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

[3] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. In *Proceedings of Language, Compilers and Tools for Embedded Systems*, 1998.

[4] J. Gonzalez. Static cache prediction based on data flow analysis for cache lines. Master's thesis, Technical University of Braunschweig, Germany, 2000.

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

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

[7] A. Hergenhan and W. Rosenstiel. Static timing analysis of embedded software on advanced processor architectures. In *Proceedings of Design, Automation and Test in Europe (DATE '00)*, pages 552–559, Paris, March 2000.

[8] M. Hill. DINERO Cache Simulator: Code, Documentation. http://www.ece.cmu.edu/ ece548/tools/dinero/src/, 1998.

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

[10] Y. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4(3):257–279, July 1999.

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

[12] R. Morgan. *Building an Optimizing Compiler*. Butterworth–Heinemann, 1998.

[13] F. Wolf and R. Ernst. Software timing and power estimation of telecom systems. ESPRIT MEDIA Published Report, Technical University of Braunschweig, Germany, 1999.

[14] F. Wolf and R. Ernst. Intervals in software execution cost analysis. In *Proceeding of the IEEE International Symposium on System Synthesis*, Madrid, Spain, 2000.

[15] W. Ye and R. Ernst. Embedded program timing analysis based on path clustering and architecture classification. In *Proceedings of ICCAD*, pages 598–604, San Jose, 1997.