ISSN:0975-9646

Stagnant Memory and Timing investigation of Embedded Systems Code

M.Shankar, Dr.M.Sridar[2], Dr.M.Rajani[3]

[1] *Associate professor & Head for PG and UG studies,*

Department of Electrical and Electronics Engineering, Kuppam Engineering College,

Kuppam, Andhra Pradesh, India,

[2]*Director of International Relations, Bharath University, Chennai, Tamilnadu, India,*

[3]*Director- Research & development. Bharath University, Chennai, India*

Abstract—Static program analysis by abstract interpretation is an efficient method to determine properties of embedded software. One example is value analysis, which determines the values stored in the processor registers. Therefore, utmost carefulness and state-of-the-art machinery have to be applied to make sure that an application meets all requirements. To do so lays in the responsibility of the system designer(s). As more software and embedded code saw use in safety-critical and avionics applications, an industry standard group developed the RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification.

Index Terms— Branch Prediction, Worst Case Execution Time, Stack, WCET

1 Introduction

Software which is embedded in aircraft to which people entrust their lives becomes safety critical and consequently must be of the highest standards. Failures of such software must be as rare as virtually non-existing during the lifetime of all aircraft concerned. Due to the high costs of aircraft, hypothetical software failures would also incur major financial losses, a further drive to require the highest quality. It is clear that in aircraft, safety is not compromise requirement one. The Army Strategic Software Improvement Program (ASSIP1) has tasked the Carnegie Mellon® Software Engineering Institute (SEI) to conduct a study of real-time, safety-critical, embedded (RTSCE) systems issues and develops recommendations for effectively dealing with those issues. This report contains the results of the first phase, an investigation into the current body of knowledge related to RTSCE system and software development, to include practices employed and emerging in the relevant United States and European commercial and government sectors. In a second phase, issues and shortfalls with the current state of RTSCE software acquisition and development practices will be identified. Recommendations for correcting those problems in the weapon systems domain will be made with emphasis on up-front tasks and considerations that will allow acquirers to position their acquisition programs properly from the start.

2 Value Analyses

We additionally have to encrypt data values stored in off-chip memory. The encryption and decryption of data values can be done by hardware. However several interesting decidable scenarios have been identified. In this section, we will see that it follows quite directly from Theorem 1 that the consistency and the implication problem for unary keys and unary inclusion constraints are decidable, even relative to structural constraints given by a regular tree language (a similar result was first shown in [1]. Other scenarios and approaches will be discussed in the next section. Because of the presence of loops, transfer and combination functions must be applied repeatedly until the system of abstract states stabilizes. Termination of this fixed point iteration is ensured on a theoretical level by the mono tonicity of transfer and combination functions and the fact that a memory location can only hold finitely many different values. Value analysis is a static analysis method based on abstract interpretation. It produces results valid for every program run and all inputs to the program. Therefore, it cannot always predict an exact value for a memory location, but determines abstract values instead that stand for sets of concrete values. There are several variants of value analysis depending on what kinds of abstract values are used. In constant propagation, an abstract value is either a single concrete value or the statement that no information about the value is known. In interval analysis, abstract values are intervals that are guaranteed to contain the exact values. Further extensions of value analysis record known equalities between otherwise unknown values, or more generally, upper and lower bounds for their differences, or even more generally, arbitrary linear constraints between values. As compiler writers are well aware, constants provide excellent optimization capability, through the well-known compiler optimization known as constant propagation and constant folding [1][10]. Such propagation consists of replacing a variable holding a constant by the constant itself. This replacement can result, for example, in branch conditions that always evaluate to false, resulting in turn in dead code that can then be eliminated. It can also enable compile-time evaluation of expressions. Such dead code resulting from constant propagation is especially common when propagating constants into subroutines through the subroutine's parameters. While the subroutine may have been designed to handle a variety of sets of parameters, a particular program may only call the subroutine with certain constant values for those parameters, resulting in much dead code in the subroutine.

3 Stack Usage Analyses

A stack overflow in an embedded DSP application generally produces a catastrophic software crash due to data corruption, lost return addresses, or both. The traditional approach to avoiding stack overflow is to perform offline testing during software development. Typically, a stack will be comfortably oversized, and the entire stack memory filled with some known data value using a code debugger. To implement this strategy, the stack buffer is set up as a circular buffer with a head and tail pointer. A pointer to memory is also needed to keep track of the top element of the memory-resident portion of the stack. Whenever a stack overflow is encountered, the bottom-most buffer-resident element is copied to memory, freeing a buffer location. Whenever an underflow is encountered, one element from memory is copied into the buffer. This technique has the appeal that the processor never moves a stack element to or from memory unless absolutely necessary, guaranteeing the minimum amount of stack traffic.

A possible embellishment of this scheme would be to have the stack manager always keep a few elements empty and at least several elements full on the stack. This management could be done using otherwise unused memory cycles, and would reduce the number of overflow and underflow pauses. Unfortunately, this embellishment is of little value on real stack machines, since they all strive to use program memory 100% of the time for fetching instructions and data, leaving no memory bandwidth left over for the stack manager to use.

Another drawback of the testing-based approach to determining stack depth is that it treats the system as a black box, providing developers with little or no feedback about how to best optimize memory usage. Static stack analysis, on the other hand, identifies the critical path through the system and also the maximum stack consumption of each function; this usually exposes obvious candidates for optimization. Using our method for statically bounding stack depth as a starting point, we have developed a novel way to automatically reduce the stack memory requirement of an embedded system. The optimization proceeds by evaluating the effect of a large number of potential program transformations in a feedback loop, applying only transformations that reduce the worst-case depth of the stack. Static analysis makes this kind of optimization feasible by rapidly providing accurate information about a program. Testing based approaches to learning about system behaviour, on the other hand, are slower and typically only explore a fraction of the possible state space. Static Analysis based approaches consist in using tools to analyze the application stack space consumption patterns and possibly compute worst case bounds prior to execution time. They usually perform some local stack consumption analysis combined with control-flow graph traversals. The bound computation is actually a small subset of a wide category of resource bounding analysis problems, focus of a lot of research activity over the years. See for an example set of publications in this area, or for a specific instance. One limitation is that a compiler cannot provide information on elements it doesn't process, such as COTS operating system services for which sources are not available or very low level routines developed in assembly language. When worst case bounds are a strong concern, not having the sources of some components is rare, however, and the stack usage in assembly routines is usually simple enough to be accounted for manually.

The compilation process may also not be able to grasp the interrupt handling bits necessary to size the worst case amount of interrupt related stack, be it for hardware interrupt or signal handlers. Interrupt handling always requires very careful design [2,3] and coding, though, so the information could at least also be provided to the framework by the user, or accounted for separately. When faced with an instruction I that is an indirect memory reference off one or more registers, we attempt to discover whether or not the register(s) used by I could be pointing into the stack.

We do this by tracing back from I in an attempt to determine the origin of the initial value of the registers in question. If we can determine that the base address being used for the pointer arithmetic is in global memory or the heap, then, from the reasoning above, we can conclude that the indirect memory reference cannot affect the stack.

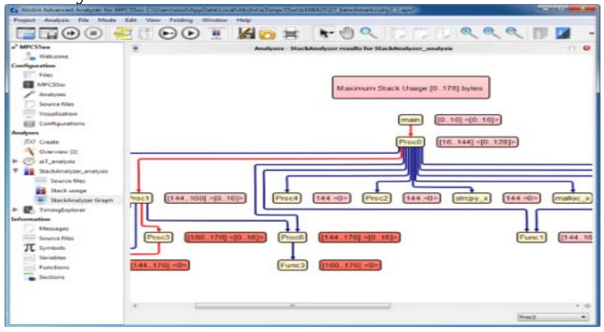


Fig1. Call graph with stack usage analysis results. All routines are annotated by their worst-case stack usage.

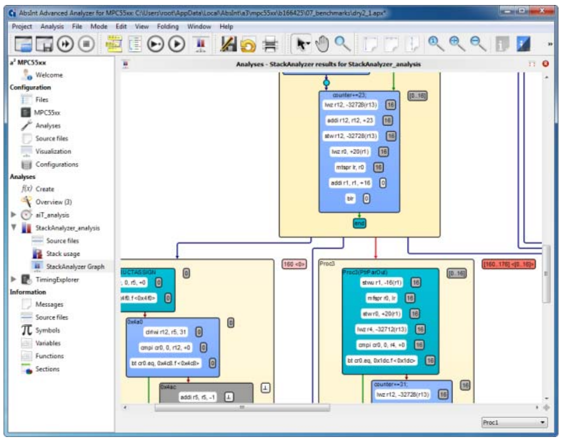


Fig2. In the control flow graph, all basic blocks and instructions are annotated by their worst-case stack usage

Stack Analyzer provides automatic tool support to calculate precise information on the stack usage. This not only reduces development effort, but also helps to prevent runtime errors due to stack overflow. Critical program sections are easily recognized thanks to color coding. The analysis results thus provide valuable feedback in optimizing the stack usage of an application. The predicted worst-case stack usages of individual tasks in a system can be used in an automated overall stack usage analysis for all tasks running on one Electronic Control Unit, as described in [7] for systems managed by an OSEK/VDX real-time operating system.

4 WCET ANALYSES: WORST-CASE EXECUTION TIME PREDICTION.

Abstract interpretation can be used to efficiently compute a safe approximation for all possible cache and pipeline states that can occur at a program point. These results can be combined with ILP (Integer Linear Programming) techniques to safely predict the worst-case execution time and a corresponding worst-case execution path. This approach can help to overcome the challenges listed in the previous sections. Abs Int's WCET tool aiT determines the WCET of a program task in several phases [4] (see Figure 1): •CFG Building decodes, i.e. identifies instructions, and reconstructs the control-flow graph (CFG) from a binary program.

•**Value Analysis** computes value ranges for registers and address ranges for instructions accessing memory.

•**Loop Bound Analysis** determines upper bounds for the number of iterations of simple loops.

•**Cache Analysis** classifies memory references as cache misses or hits.

•**Pipeline Analysis** predicts the behaviour of the program on the processor pipeline.

Hard real-time systems need methods to determine upper bounds for their execution times, usually called worst-case execution times, (WCET). Based on these bounds, a schedulability analysis can check whether the underlying hardware is fast enough to execute the system's task such that they all finish before their deadlines. This problem is nontrivial because performance-enhancing architectural features such as caches, pipelines, and branch prediction introduce "local non-determinism" into the processor behaviour; local inspection of the program cannot determine what the contribution of an instruction to the program's overall execution time is. The execution history determines whether the instruction's memory accesses hit or miss the cache, whether the pipeline units needed by the instruction are occupied or not, and whether branch prediction is correct or not.

4.1 Structure of WCET Computation

We first present some more details about the structure of the program. It consists of 24 uninterruptible tasks that are activated one-by-one by a real time clock in a fixed schedule: task 1 to task 24, then task 1 again, and so on until the electrical power of the aircraft is switched off. This time-triggered scheduling method requires that the WCET of each task must be less than the period of the real-time clock. The call graph of each task is basically organized in three layers. The first layer contains 4 calls to so-called sequencers, which for each task are selected from a list of 38 possibilities. These sequencers allow for the activation of pieces of code at different rates, i.e., 1 over 2 ticks, 4 ticks, 8 ticks or 24 ticks. Still in this highest layer, some system routines are called before and after the four sequencer calls. The second layer consists of the routines containing the actual operation code composed of "calls" to code macros, which form the basic components referred to in section 3. The third layer consists of the input/output routines called by some of the basic components present in the second layer. The major part of the factors affecting the WCET (conditions, loop bounds, pointers, etc) is found automatically by aiT, either by code inspection or from the annotations describing the configuration table. Yet some factors are outside aiT's knowledge and capacities [4,5,6], and annotations have to be provided to bound the analysis and achieve a result. These factors are lower and upper bounds on input data, static data from previous task activations, or data provided by devices outside of processor knowledge (DMA for example). For these, maximum loop iterations, values read from memory, branch exclusions, etc, have to be specified to aiT.

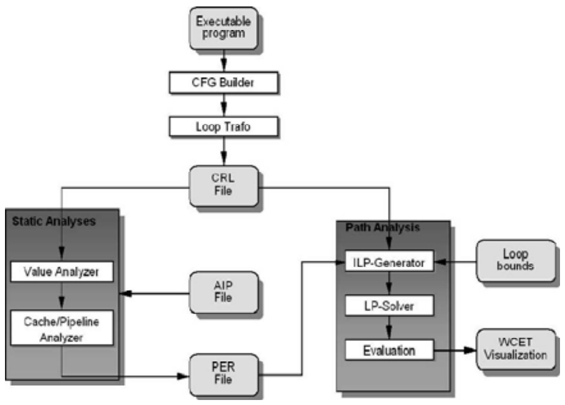


Figure3. Phases of WCET computation

Cache Analysis uses the results of value analysis to predict the behaviour of the (data) cache. The results of cache analysis are used within pipeline analysis allowing the prediction of pipeline stalls due to cache misses. The combined results of the cache and pipeline analyses are used to compute the execution times of program paths. Separating the WCET determination into several phases makes it possible to use different methods tailored to the subtasks. Value analysis, cache analysis, and pipeline analysis are done by abstract interpretation [2], a semantics-based method for static program analysis. Integer linear programming is used for path analysis.

4.2 Reconstruction of the Control Flow from Binary Programs

The starting point of our analysis framework is a binary program and additional user-provided information about numbers of loop iterations, upper bounds for recursion, etc. In the first step a parser reads the compiler output and reconstructs the control flow. This requires some knowledge about the underlying hardware, e.g., which instructions represent branches or calls. The reconstructed control flow is annotated with the information needed by subsequent analyses and then translated into CRL (Control Flow Representation Language, a human-readable intermediate format designed to simplify analyses and optimizations at the executable/assembly level). This annotated control flow graph serves as the input for micro architecture analyses.

Logical Framework Analysis or the Logical Framework Approach (LFA) is an analytical process for structuring and systematizing the analysis of a project or programme idea. It is useful to distinguish between LFA, which is a process involving stakeholder analysis, problem analysis, objective setting and strategy selection – and the logical framework matrix, often called [7] the log frame, which documents the product of the LFA process. Intervention logic: the description of the project according to its hierarchy of objectives – the strategy underlying the project Vision: the desired state or ultimate condition that a project is working to achieve and to which the project contributes Goal: a desired impact of a project – ambitious yet realistic; direct benefits to the conservation target; the project is held responsible for achieving its stated goal(s) Objective: a desired accomplishment or outcome of a project, such as the reduction of a critical threat – the actual change in a problem targeted by the project Results: the tangible products or services delivered by the project Strategic activity: A specific action or set of tasks to reach one or more results (or objectives); activities can be added as a fifth row under results, but this is no longer current practice among most donors, and activities do not need indicators Indicator: a measurable entity related to a specific information need, such as the status of a target/factor, change in a threat, or progress toward an objective; a good indicator meets the criteria of being measurable, precise, consistent, and sensitive.

4.3 Data flow analysis

The aim of the data flow analysis phase is to transform the low-level intermediate representation into a higher-level representation that resembles a HLL statement. It is therefore necessary to eliminate the concept of condition codes (or flags) and registers, as these concepts do not exist in high-level languages, and to introduce the concept of expressions, as these can be used in any HLL program. For this purpose, the technology of compiler optimization has been appropriated. The first analysis is concerned with condition codes. Some condition codes are used only by hand-crafted assembly code instructions, and thus are not translatable to a high level representation. Therefore, condition codes are classified in two groups: HLCC which is the set of condition codes that are likely to have been generated by a compiler (e.g. overflow, carry), and NHLCC which is the set of condition codes that are likely to have been generated by assembly code (e.g. trap, interrupt). The control flow analyser structures the control flow graph into generic high-level control structures that are available in most languages. These are conditional (if then [.else]), multi way branch (case), and different types of loops (while (), repeat until, and endless loop). Different methods have been specified in the literature to structure graphs, most of them dealing with the elimination of go to statements from the graph, by the introduction of new variables16,17, code replication18,19,20 or the use of multilevel exit21,22. Both the introduction of new variables and code replication modify the apparent semantics of the program, and is therefore not desirable when decompiling binary programs, given that we want to decompile the code ‘as is’. The use of multilevel exit statements is not supported by commonly used languages (e.g. Pascal, C), and thus cannot be part of the generic set of high-level control constructs that can be generated. We developed an algorithm that structures the graph into the set of generic high-level control structures, and, whenever it determines that a particular sub graph is not one of the generic constructs, it uses a go to. Note that the minimum number of go to is always used.

4.4 Value Analysis

On the hardware front, the rate of improvement in microprocessor CPU speed continues to exceed the rate of improvement in DRAM memory speed, producing an increasing gap between processor and memory performance [Patterson and et al. 1997]. Multiple levels of cache hierarchy have been introduced to alleviate the CPU-memory performance gap. However, it is also vital to optimize memory usage to achieve better performance. Furthermore, in recent years, power and energy consumption have become critical design issues for both high-end systems and embedded devices [8,9,10]. A significant source of processor energy consumption is on-chip cache [Kamble and Ghose 1997; Mudge 2001], where memory loads and stores dissipate energy. Optimization of memory accesses thus can also improve application energy efficiency.

4.5 Loop Bound Analysis

Upper bounds on the number of loop iterations are needed in order to derive a finite WCET estimate at all. Similarly, recursion depth must also be bounded. Due to the halting problem, no automatic method for loop bounds analysis can give an exact answer for all loops. Thus, WCET analysis tools provide means to give loop iteration bounds manually [5, 6, 17]. However, this is often laborious, and a source of possible errors. Although necessarily incomplete, an automatic loop bounds analysis can still be useful to reduce the manual work by bounding most of the commonly occurring loops. A common approach is to identify loop counters, and then determine (or bound) their start values, increment (decrement), and highest (or lowest) possible value. From this information, an upper bound for the iteration count can be obtained. Whalley et al. [12] use data flow analysis and specialized algorithms to calculate loop bounds for both single and some special types of nested, triangular loops. This approach is quite syntactical and will fail for loops which do not fit the patterns. The loop-bound analysis of the Bound-T tool [17] estimates range and increment for loop counters using Presburger arithmetics, and the latest loop bound analysis of the aiT tool [4] decides start values by an interval-based AI and the possible increments by a data flow analysis. These methods have in common that they only work for well-structured loops with a proper nesting, and where loop counters are updated using addition or subtraction only.

4.5 Cache Analysis

The rate at which the processor can execute instructions is limited by the memory cycle time. This limitation has in fact been a significant problem because of the persistent mismatch between processor and main memory speeds. Caches—which are relatively small high speed memories—have been introduced in order to hold the contents of most recently used data of main memory and to exploit the phenomenon of locality of reference (see Hennessy and Patterson, 1990). The advantage of a cache is to improve the average access time for data located in main memory.

4.6 Pipeline Analysis

Pipeline analysis models the pipeline behaviour to determine execution times for sequential flows (basic blocks) of instructions, as done in [11]. It takes into account the current pipeline state(s), in particular resource occupancies, contents of prefetch queues, grouping of instructions, and classification of memory references by cache analysis. The result is an execution time for each basic block in each distinguished execution context. Like value and cache analysis, pipeline analysis is based on the framework of abstract interpretation. Pipeline analysis of a basic block starts with a set of pipeline states determined by the predecessors of the block and lets this set evolve from instruction to instruction by a kind of cycle-wise simulation of machine instructions. In contrast to a real simulation, the abstract execution on the instruction level is in general non-deterministic since information determining the evolution of the execution state is missing, e.g., due to non-predictable cache contents.

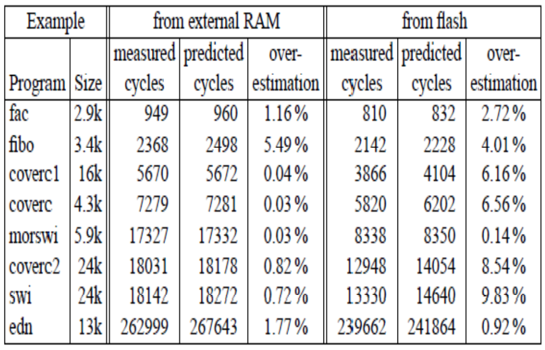
5. PATH ANALYSIS

A main issue in WCET analysis is to avoid pessimism while being safe in timing evaluation. Ideally, WCET estimation method should, given an input program, produce a tight estimate of the upper-bound of the actual WCET. But first, we need a timing model of the hardware platform. Indeed, such micro-architecture modelling for low-level analysis is non-trivial and it is almost impossible to achieve exact WCET estimates in CPU cycles. Second, it is crucial to estimate accurately bounds for loops and eliminate infeasible paths from bound calculation, especially in the presence of nested loops. This can be partially addressed by requiring user-provided path annotations and loop bound information. Apart from considerable effort and error-proneness, sometimes the user may not actually know such information. A more attractive solution is to automatically detect infeasible paths and derive loop bounds through static path analysis methods [2, 12, 16, 17].

6 PRECISION OF AIT

Since the real WCET is not known for typical real-life applications, statements about the precision of aiT are hard to obtain. For an automotive application running on MPC 555, one of AbsInt's customers has observed an overestimation of 5–10% when comparing aiT's results and the highest execution times observed in a series of measurements (which may have missed the real WCET). For an avionics application running on MPC 755, Airbus has noted that aiT's WCET for a task typically is about 25% higher than some measured execution times for the same task, the real but non-calculable WCET being in between. Measurements at AbsInt have indicated overestimations ranging from 0% (cycle-exact prediction) till 10% for a set of small programs running on M32C, TMS320C33, and C16x/ST10. Table 1 shows the results for C166. The analysis times were moderate—a few seconds till about 3 minutes for edn.

PRECISION OF AIT FOR SOME C166 PROGRAMS



Example		from external RAM			from flash		
Program	Size	measured cycles	predicted cycles	over-estimation	measured cycles	predicted cycles	over-estimation
fac	2.9k	949	960	1.16%	810	832	2.72%
fibo	3.4k	2368	2498	5.49%	2142	2228	4.01%
coverc1	16k	5670	5672	0.04%	3866	4104	6.16%
coverc	4.3k	7279	7281	0.03%	5820	6202	6.56%
morswi	5.9k	17327	17332	0.03%	8338	8350	0.14%
coverc2	24k	18031	18178	0.82%	12948	14054	8.54%
swi	24k	18142	18272	0.72%	13330	14640	9.83%
edn	13k	262999	267643	1.77%	239662	241864	0.92%

7. CONCLUSION

In this paper, we presented an abstract interpretation based static analysis framework for analyzing hard-codedness of pointer variables in embedded assembly code. Our results show that static analysis based approaches is viable in industrial settings for checking for coding standards compliance. Code compliance checking is critical for code reuse and COTS compatibility in applications. A complete analyzer has been developed for pointer hard-codedness analysis and shown to run successfully on code samples taken from Texas Instruments' DSP code suite. The prototype system is currently being refined to provide more accurate results in presence of global pointers and mutually recursive functions. Future work also includes extending the system to handle rules 1, 2, and 4 through 6 [11] laid out by TI. Note that the analyses needed for rule numbers 4 and 6 are very similar to hard-codedness analysis. Similarly rules 2 and 5 require analysis that determines if a binary code is entrant.

References

[1] Samuel Z. Guyer, Calvin Lin. Client-Driven Pointer Analysis. Static Analysis Symposium. 2003. Springer LNCS 2694. pp. 214-236.

[2] S. Adams, T. Ball, et al. Speeding Up Dataflow Analysis Using Flow-Insensitive Pointer Analysis. SAS 2002. pp. 230-246

[3] Donglin Liang, Mary Jean Harrold. Efficient Computation of Parameterized Pointer Information for Interprocedural Analyses. SAS 2001. Springer LNCS 2126. pp. 279-29.

[4] D. Brylow, N. Damgaard, J. Palsberg, Static Checking of Interrupt-driven Software. International Conference on Software Engineering. 2001.

[5] W. Amme, P. Braun, E. Zehendner, F. Thomasset. Data Dependence Analysis of Assembly Code. Proc. PACT 1998.

[6] M. Fernandez and R. Espasa. Speculative alias analysis for executable code. Proc. PACT 2002.

[7] J. Bergeron, M. Debbabi, M.M. Erhioui, B. Ktari. Static Analysis of Binary Code to Isolate Malicious Behaviors. IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 1999. Palo Alto, California

[8] Saumya Debray, Robert Muth, Matthew Weippert Alias analysis of executable code. POPL'98.

[9] Thomas Lundquist and Per Stenstr¨om. Timing anomalies in dynamically scheduled microprocessors. In Proceedings of the 20th IEEE Real-Time Systems Symposium, December 1999.

[10] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In Kai Koskimies, editor, Proceedings of the International Conference on Compiler Construction (CC'98), volume 1383 of Lecture Notes in Computer Science, pages 80–94. Springer-Verlag, March /April 1998.

[11] J¨orn Schneider and Christian Ferdinand. Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation. In Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems, volume 34, pages 35–44, May 1999.

[12] John A. Stankovic. Real-Time and Embedded Systems. ACM 50th Anniversary Report on Real-Time Computing Research, 1996. http://www-ccs.cs.umass.edu/sdcr/rt.ps.

[13] Henrik Theiling. Extracting Safe and Precise Control Flow from Binaries. In Proceedings of the 7th Conference on Real-Time Computing Systems and Applications, Cheju Island, South Korea, 2000.

[14] Henrik Theiling. Generating Decision Trees for Decoding Binaries. In Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems, pages 112–120, Snowbird, Utah, USA, June 2001.

[15] Henrik Theiling. ILP-based interprocedural path analysis. In Alberto L. Sangiovanni-Vincentelli and Joseph Sifakis, editors, Proceedings of EMSOFT 2002, Second International Conference on Embedded Software, volume 2491 of Lecture Notes in Computer Science, pages 349–363. Springer-Verlag, 2002.