Deadline Analysis of Interrupt-Driven Software

Dennis Brylow[*] Jens Palsberg[†]

August 2004

Abstract

Real-time, reactive, and embedded systems are increasingly used throughout society (e.g., flight control, railway signaling, vehicle management, medical devices, and many others). For real-time, interrupt-driven software, timely interrupt handling is part of correctness. It is vital for software verification in such systems to check that all specified deadlines for interrupt handling will be met. Such verification is a daunting task because of the large number of different possible interrupt arrival scenarios. For example, for a Z86-based microcontroller, there can be up to six interrupt sources and each interrupt can arrive during any clock cycle. Verification of such systems has traditionally relied upon lengthy and tedious testing; even under the best of circumstances, testing is likely to cover only a fraction of the state space in interrupt-driven systems.

This paper presents the Zilog Architecture Resource Bounding Infrastructure (ZARBI), a tool for deadline analysis of interrupt-driven Z86-based software. The main idea is to use static analysis to significantly decrease the required testing effort by automatically identifying and isolating the segments of code that need the most testing. Our tool combines multi-resolution static analysis and testing oracles in such a way that only the oracles need to be verified by testing. Each oracle specifies the worst-case execution time from one program point to another, which is then used by the static analysis to improve precision. For six commercial microcontroller systems, our experiments show that a moderate number of testing oracles are sufficient to do precise deadline analysis.

keywords: Real time, multi-resolution static analysis, testing oracles

1 Introduction

Real-time systems have become pervasive in the world. Commerce, health care, transportation, and telecommunication all rely increasingly on real-time sensing and control. Particularly for applications in areas that are a matter of life and death, the correctness of real-time software is of paramount importance.

[*] Purdue University, Dept of Computer Science, W Lafayette, IN 47907, USA, brylow@purdue.edu, phone: 765–494–7843.

[†] University of California Los Angeles, Dept of Computer Science, Los Angeles, CA 90095-1596, USA, palsberg@ucla.edu, phone: 310–825–6320.

1.1 Background

Correctness of real-time software can be thought of as having two parts. The first issue is correctness of input-output behavior, and the second is timeliness of that behavior. Verification and validation of input-output behavior has been widely studied; there are now many static-checking tools available, including type checkers [12], bytecode verifiers [31], and model checkers [13], as well as numerous tools for supporting the test process. Verification of timing properties is more difficult, but steady progress has been made toward understanding the foundations of checking the timing properties of real-time software in recent years [6, 5]. However, major open issues still remain. These issues are due to the low-level nature of real-time systems, with most still implemented either in assembly language or at lower levels, such as FPGAs or custom-built ASICs. Even when real-time software is written in a higher-level language such as C, it is desirable to check the real-time properties of the compiled code because it can be difficult to predict the effects of the compiler. Most previous work on analysis of assembly code [58, 7, 47] is not concerned with timing properties.

The goal in this project is to provide tool support for checking timing properties of real-time assembly code. This work focuses on interrupt-driven software, where a signal from a source outside the direct control of the software can cause computation to be interrupted by control being transferred to an interrupt handler. Typical interrupts in such systems occur because new sensor data is available, a signal pulse arrives at the controller, an internal timer goes off, or for many other reasons. The specification of an interrupt-driven system will usually list deadlines for the handling of each type of interrupt. It is part of the correctness of the system that all deadlines are met. Reasoning about the timing behavior of interrupt-driven software is complicated because interrupts can be enabled and disabled by the software itself, an interrupt handler can be interrupted, and interrupts can arrive in a myriad of different scenarios. It is critical to know whether an interrupt arrives at a point where it is enabled and can be handled right away, or whether it arrives 50 clock cycles later, when the system has just disabled interrupt handling and will be doing other work for the next two million clock cycles. In summary, the interrupts and the main computation can interact in potentially bizarre ways. Programs in such a style can be found, for example, in the area of sensor networks [23] where a sensor node may not have space for a sophisticated scheduler and analysis of interrupts is known to be difficult [47, 29].

Tool support for real-time interrupt-driven software strives to answer the following question.

Deadline Analysis: Will every interrupt be handled before its deadline?

One can approach this question in a testing-based manner: try a suite of interrupt schedules and measure whether all deadlines are met. Developing a good suite of interrupt schedules is a difficult problem because of the fine granularity of the timing domain. Even if a clock cycle is as long as one microsecond, it is very difficult to engineer or discover interrupt schedules that lead to any reasonable coverage of the program. Statement coverage would be easy in this setting, but is not a useful coverage criteria because it does not take into account the interplay of different interrupts and the times when they occur. Branch coverage is more accurate but far more expensive; at every program point where an interrupt is enabled, there is an implicit branch to the handler. Covering all branches can therefore be an intractable task. In summary, the problem with a test-based approach is that it is difficult to test a sufficiently wide variety of schedules to gain confidence in the software.

An alternative is a static-analysis-based approach to deadline verification. A good illustration of the difficulties faced by that approach is given in our earlier paper with Damgaard [9], which showed that for six commercial microcontrollers the maximal stack size for interrupt-driven assembly code could be estimated successfully by a static analysis. The experiments of that paper also illustrated that static analysis of timing properties cannot work without information about the behavior of external devices. For example, if the code uses a loop to busy-wait on a new value from a port, static analysis will view it as an infinite loop, even if the programmer knows that an external device will deliver a new value every 100 milliseconds. Once the static analysis has detected an infinite loop on the path from A to B, it will determine that if an interrupt occurs when the execution is at program point A and the handler for the interrupt has exit point B, the handling may never terminate, let alone meet its deadline. In summary, the static analysis approach of [9] predictably failed to perform useful deadline analysis.

Our thesis is that we should *combine* static analysis and testing. In practical terms, the fundamental challenge is:

Challenge: Can static analysis significantly decrease the required testing effort?

There are previous success stories of combining static analysis and testing. For example, in the area of regression testing, rather than re-running the software on the whole test suite every time a change has been made, one can use static analysis to conservatively estimate which test inputs must be tried again [26]. In the interrupt-driven setting, static analysis can reduce the required testing effort, allowing the testing effort to be more *focused*, which is exactly what is desired with a combined static-analysis/testing approach to deadline analysis.

Our approach uses test oracles [49] for certain worst-case execution time (WCET) questions that cannot possibly or easily be answered by static analysis. The oracles assert to the static analysis that if execution reaches program point A, then it will reach program point B at most t microseconds later. When A and B are close, then a much smaller testing effort is required to verify such an oracle than to do the entire deadline analysis. Moreover, if more than one oracle is needed for a program, the work of validating the different oracles can be done in parallel. Our goal is to combine static analysis with timing oracles to improve the precision of the deadline analysis.

Deadline analysis cannot be performed without WCET analysis. However, most research on deadline analysis assumes that WCET analysis has already been successfully completed, and most published papers on WCET analysis do not consider the needs of deadline analysis. Many papers in this area concentrate on estimating the execution time from one program point to another, usually from start to finish, sometimes even focusing on a particular input, and they rarely handle interrupts [42, 53, 18, 20, 8, 56, 14]. Deadline analysis is more complicated than simple WCET analysis because the interrupts can occur at any time and their handlers can be enabled or disabled at any program point. In deadline analysis, the starting point for the analysis is not given. It is a task of the analysis to identify the worst-case program point at which an interrupt can occur and then estimate the WCET to the exit point of the handler for that interrupt.

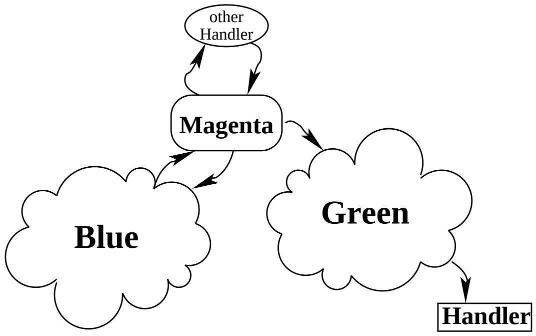


Figure 1: Coloring a Flow Graph

In summary, deadline analysis for interrupt-driven assembly code remains a difficult and little-studied problem.

1.2 Experimental Results

We have designed and implemented ZARBI (Zilog Architecture Resource Bounding Infrastructure), a tool for integrated deadline and WCET analysis of interrupt-driven assembly code. In slogan form:

$$\text{deadline analysis} \quad = \quad \text{static analysis} \quad + \quad \text{testing oracles.}$$

For six commercial microcontroller programs, each on the order of 1000 lines of code, we found that at most 17 oracles were sufficient for each program. In our experience, an expert user can add the oracles in less than an hour, in an interactive fashion, until the deadline analysis is complete.

Our tool uses a multi-resolution analysis, which allows it to explore difficult segments of the control flow graph in sufficient depth to bound the latency while staving off the intractable complexity that would arise from using such fine-grained analysis over the whole program.

Our static analysis proceeds by building and coloring a flow graph. Each node is given one of five colors: *Green*, *Magenta*, *Blue*, *Yellow*, and *Red*. Intuitively, *Green* means that the time bound can be found, *Magenta* means that starvation is possible, *Blue* means that starvation is possible later in the computation, *Yellow* means that the analysis thinks that the time bound might not be calculable, and *Red* means that the analysis is certain that there is no time bound. For our test suite, no red nodes were found, we were able to eliminate all yellow nodes by adding oracles, and we observed that very few nodes were magenta.

Figure 1 illustrates a flow graph at the time the deadline analysis is complete, that is, when all yellow nodes have been eliminated. Notice that "other Handler" can starve an interrupt that is to be handled by "Handler."

Our tool is intended to be used as part of a three step process. For a given interrupt, (1) add oracles until all nodes are green, magenta, or blue, (2) use simulation and testing to find a WCET for the magenta clouds, and (3) combine the WCET's from the green, blue, and magenta clouds to compute the WCET for handling the interrupt.

The remaining sections of the paper proceed according to the following outline: Section 2 covers related work. Section 3 reviews control flow graph terminology and concepts that are used throughout the technical sections of the paper. Section 4 explains the coloring of control flow graphs for deadline analysis purposes, and section 5 shows how testing oracles have been integrated into our tool. Section 6 describes the multi-resolution analysis we have developed for deadline analysis. Section 7 details implementation issues encountered while implementing our prototype, and section 8 presents our results and walks the reader through a session with ZARBI. Section 9 summarizes and closes.

2 Related Work

In the general case, the problem of bounding stack sizes and maximum execution times is equivalent to the halting problem [48]; it is a basic theorem of computer science that these questions are undecidable. Much work has been done on tools that operate on decidable subsets of programming languages, for example, Berkeley Packet Filters [35], or Agere Systems' C-NP language [1] for programming network processors, which do not allow backward branching.

Most research in the area of calculating real-time software resource bounds stems from Puschner and Koza's work [45], which uses the following conditions to guarantee decidability: no asynchronous interrupts, no recursion, no indirect calls, no `goto` instructions, and only strictly bounded loops.

In the 1990's, researchers worked to relax several of these restrictions, with a variety of trade-offs. However, despite the fact that asynchronous interrupts are the most salient feature of actual real-time systems, they remain the least researched topic on the above list.

2.1 The False Path Problem

Altenbernd identified that a key issue in accurate worst-case execution time (WCET) analysis is the False Path Problem [4]. In constructing a control-flow graph, the abstraction often contains paths that cannot actually take place in a real program execution. In order to calculate tight bounds on execution time, the algorithm must search for the longest executable path in the graph, rather than the longest structural path in the graph. This is equivalent to an NP-complete problem that exists in hardware design; finding the longest executable path in a network of logic gates is substantially more difficult than finding the longest structurally connected path [33]. Altenbernd used symbolic execution to track possible values of key conditional variables, thereby pruning unrealizable paths out of the control-flow graph. This is essentially the same technique used by our tool to prune away a substantial number of unrealizable interrupt handler paths from the control-flow graphs.

2.2 Testing Oracles

There are previous success stories of combining static analysis and testing. For example, in the area of regression testing, rather than re-running the software on the whole test suite every time a change has been made, one can use static analysis to conservatively estimate which test inputs must be tried again [26]. In our setting, static analysis can reduce the required testing effort, allowing the testing effort to be more *focused*, which is exactly what we desire to achieve with a combined static-analysis/testing approach to deadline analysis.

Our approach uses test oracles [49] for certain worst-case execution time (WCET) questions that cannot possibly or easily be answered by static analysis. The oracles assert to the static analysis that if execution reaches program point A, then it will reach program point B at most t microseconds later. When A and B are close, then a much smaller testing effort is required to verify such an oracle than to do the entire deadline analysis. Moreover, if more than one oracle is needed for a program, the work of validating the different oracles can be done in parallel. Our goal is to combine static analysis with timing oracles to improve the precision of the deadline analysis.

2.3 Worst-Case Execution Time Analysis

Deadline analysis cannot be performed without WCET analysis. However, most research on deadline analysis assumes that WCET analysis has already been successfully performed, and most published papers on WCET analysis do not consider the needs of deadline analysis. Many papers in this area concentrate on estimating the execution time from one program point to another, usually from start to finish, sometimes even focusing on a particular input, and they rarely handle interrupts [42, 53, 18, 20, 8, 56, 14]. Deadline analysis is more complicated than simple WCET analysis because the interrupts can occur at any time and their handlers can be enabled or disabled at any program point. In deadline analysis, the starting point for the analysis is not given. It is a task of the analysis to identify the worst-case program point at which an interrupt can occur and then estimate the WCET to the exit point of the handler for that interrupt.

Work in automatic detection of induction variables [36], and bounding of *unnatural* loops in low-level languages [27] is applicable to loops present in the commercial microcontroller systems examined later in this paper. Healy and Whalley's approach [28] concentrates on the branch instructions themselves. By searching backward to find all of the assignments that influence registers used in the branch comparison, they are able to classify all jumps as one of *unknown*, *fall-through*, or *jump*. The search continues until all registers in the expression can be replaced by immediate values, or a control-flow merge point is encountered. This intra-procedural analysis allows tighter bounds to be calculated for many loops.

2.4 Call Graphs and Model Checking

A static analysis of assembly code may attempt to approximate the values in specific registers or on the stack. This problem is closely related to the problems of call-graph construction and points-to analysis for object-oriented programs. Accurate, scalable analyses for these purposes exist in the programming languages community [41, 55].

The FLAVERS system at University of Massachusetts, (FLow Analysis for VERifying Specifications), is a flexible framework for flow analysis of concurrent programs [16, 37]. FLAVERS has even been extended to analyze infinite executions [38], which are common in embedded systems. However, the FLAVERS system has a much higher-level abstraction of concurrent tasks; separate tasks do not have completely shared stack and data registers. Such a high-level analysis thrives on a more rigidly specified interface between tasks than can exist at the Z86 microcontroller level.

The stack-size checking algorithm in ZARBI can be seen as a demand-driven version of an algorithm for model checking of pushdown systems like Podelski [44]. The algorithm presented in our work differs from Podelski [44] in that it generates edges on demand, thereby ensuring that many unreachable nodes are automatically pruned away. This demand-driven quality, combined with tight approximation of feasible IMR values, prevents the exponential state-space explosion that would occur in more naïve analyses.

Maximum execution time is formulated as a graph theoretic problem in Puschner and Schedl [46], using T-graphs and ILP solvers..

Like Brylow et al. [9], Wegener and Mueller [57] shows that static analysis and evolutionary testing can be used in concert to seek both upper and lower bounds on worst-case execution time.

2.5 Tools

The ASTEC group [19] represents control flow using a basic unit called a *scope*, which is intuitively a looping construct. All scopes have an iteration counts associated with them; non-looping code is a scope with zero or one iteration. Scopes are assembled into a *scope tree*, which implicitly represents all possible control flow in the program. Scopes are a very general concept, to which a wide variety of *execution facts* can be attached, including flow information facts [18] to describe feasible execution paths, or facts about low-level factors like pipeline effects on the execution time [20]. Scope trees are processed into a system of constraints using an implicit path enumeration technique (IPET) analysis to determine the maximum execution count for each point in the program. The ASTEC infrastructure now includes support for flow analysis of C programs [24].

The USES group has used abstract interpretation [15, 39] and ILP solvers to extensively model the Motorola "ColdFire" MCF 5307 processor [21]. Their modular architecture breaks down the overall WCET problem into smaller parts: a value analysis approximates possible addresses of memory accesses; a cache analysis characterizes all memory accesses as hits or possible misses; a pipeline analysis takes into account the speedup caused by subsequent instructions passing through the pipeline in succession; a final path analysis calculates the WCET of the program. Each analysis can make use of information provided by the previous analysis in the chain. The USES group's tool has been applied to test programs supplied by AIRBUS [21].

Commercial ILP solvers like CPLEX[30] and `lp_solve`[40] have been employed to analyze advanced processor features like cache and pipeline analysis[3, 22], and branch prediction[34].

2.6 Summary of Related Work

Much work has been done on timing schema for high-level languages, and on mitigating the timing effects of pipelines and caches in modern processors. Symbolic execution and implicit path merging are among several techniques intended to eliminate false paths in representative control-flow graphs in order to keep static analysis tractable in size. Model checking and type system advances have been used to verify many useful software properties. Nevertheless, previous work in the area of bounding resources for real-time software can be separated into two categories:

- Work that ignores preemptive interrupts altogether, and
- Work that assumes interrupt handlers are trivially isolatable from the main process.

All of the real-time systems examined in our work have interrupt handlers heavily integrated with the main program; they share the same system stack, have no memory protection, and in many cases affect control flow within the main program. Prior research does not attempt analysis of interrupt handlers as an integral part of the real-time system, and thus cannot provide useful bounds on interrupt-driven systems. Furthermore, for most prior work, the exponential increase in state-space that occurs when taking interrupt-handler control-flow into account would make analysis largely intractable.

This paper presents techniques for analysis of interrupt-driven programs that mitigate much of the exponential increase in state-space during analysis.

3 Control Flow Graphs

This section reviews common control flow graph terminology and concepts that will be used throughout the remainder of the paper. Readers familiar with this material can safely skip ahead to the next section. As a preview, we will be using control flow graphs in which nodes represent program states, edges represent possible transitions between states, and edge weights can represent either timing information or changes in stack height; stack analysis graphs can have zero- and negative-weighted edges.

A *control flow graph* [2] is an abstraction of program states and the transitions between them. Details and examples of control flow graph construction are given in later sections.

Control flow graphs (hereafter abbreviated as *CFG's*) are a special case of the general graph data structure. A graph G is defined as the tuple $\langle V, E \rangle$, consisting of a finite set of vertices V and edges $E \subseteq V \times V$. A vertex is also sometimes called a *node*. A control flow graph is a tuple $\langle G, w, terminus \rangle$, where w is a weight function that maps edges $e \in E$ to integers ($w : E \mapsto \mathbb{Z}$) and *terminus* is the designated vertex ($terminus \in V$) to be the starting or ending point of a search in the CFG.

The CFG is a *digraph* [50], meaning that all edges $e \in E$ are *directed*, or one-way; the first vertex in e is the *source*, and the second vertex is the *destination*. Let $A(v)$ be the set of edges $e \in E$ such that v is the destination vertex for e. Let $\Omega(v)$ be the set of edges $e \in E$ such that v is the source vertex for e. $A(v)$ is vertex v's *incoming* edge set, and $\Omega(v)$ is v's *outgoing* edge set.

The vertices in a CFG are an abstraction of the possible states in the corresponding program. As such, each vertex may be associated with a wide range of state information. In the context of our analysis, a CFG vertex will consist of the triplet $\langle PC, \sigma, IMR\rangle$, where PC is the program counter value associated with program point, σ is a call string suffix (explained below), and IMR is an approximation of the interrupt mask register's value. (The purpose of the IMR will be explained in the next section.)

Call string suffixes are calculated using stacks. In general terms, a *stack* is a last-in, first-out data structure. The stack has at least two operations defined, *push* and *pop*. An element x pushed onto a stack σ results in a new stack, $x\sigma$. The pop operation on a stack $x\sigma$ returns element x and stack σ. Let the pop operation be undefined for an empty stack, written "$\{\}$". A *call site* [2] is a program point from which another program point (subroutine or function) is invoked. A CFG that distinguishes program states within a function by distinct call sites is *context sensitive* [39]. Context sensitivity can be expressed quite naturally as a stack of call sites at each node in the CFG. A stack of call sites is also known in the literature as a *call string* [51, 39]. The CFG's constructed in our analysis contain call strings or *call string suffixes* at each vertex.

3.1 Control Flow Paths

Resource-bounding algorithms deal extensively with paths in CFG's. A *control flow path*, or *path*, π is a sequence of vertices $v_0, ..., v_k$ such that $\forall i \in \{0, ..., k-1\} : \langle v_i, v_{i+1}\rangle \in E$. A *simple path* is a path in which each v_i in π is distinct. A *cycle* [50] consists of a simple path from v_0 to v_k, with an additional edge from v_k back to v_0. A vertex v_k is *reachable* from vertex v_0 if there exists a path from v_0 to v_k. A vertex v_0 is *upstream* of v_k if there exists a path from v_0 to v_k, but not vice-versa.

The resources to be analyzed in a CFG can be represented as edge weights. The weight function w maps each edge to an integer cost. G is therefore a *weighted digraph*, or *network* [50]. Every path π has a *path weight* or *cost* $C(\pi) = \sum_{i\in\{0,...,k-1\}} w(v_i, v_{i+1})$. Let a *null path* be a path in which $\forall i \in \{0, ..., k-1\} : w(v_i, v_{i+1}) = 0$.

With call string suffixes present at every vertex, the CFG contains a notion of *valid* or *realizable* paths in the CFG. Realizable paths $\pi_{real} \in G$ are those in which the sequence of program states corresponding to vertices along π_{real} preserve the procedure call semantics of the original program. That is, for all π_{real} outgoing from vertex v_{call}, π_{real} returns from the procedure subgraph to call site v_{call}, rather than some other call site.

The *longest path* [50] in the graph is defined as the path with the largest cost, which is not necessarily the path with the largest number of edges. An analysis for bounding stack size or interrupt latency in a CFG is at its core a longest realizable path problem. The chief difficulty in finding the longest realizable path in these graphs is in first constructing a suitable CFG that expresses external information not normally available to control flow analysis.

3.2 Control Flow Cycles

Many of the algorithmic details of resource bound analysis in this paper deal with the different types of cycles in CFG's. A *negative cycle* refers to a cycle π in which $C(\pi) < 0$.

A cycle is said to be *positive* if $C(\pi) > 0$. A *zero-weight cycle* is one in which $C(\pi) = 0$. A zero-weight cycle which is also a null path is a *null cycle*.

The longest path in a graph is undefined if the graph contains negative cycles. While it is possible to construct actual programs that result in negative cycles in CFG's, (when edge weights represent change in stack height, for example,) such programs are outside the scope of our analysis. Negative cycles can be detected in $O(V^3)$ using Floyd's Algorithm [50].

The longest path in a CFG is not defined for graphs with positive cycles. (At least, it is not defined in the context of this paper.) If a positive cycle exists in the graph, a path can become arbitrarily long by passing through the cycle multiple times. A graph with neither negative nor positive cycles is *bounded*. Given a graph G that has no negative edges, positive cycles can be checked for by a bounded depth-first search, in which a graph is not bounded if the cost of a path exceeds a given boundary, m. For stack size analysis it is assumed that there is a known bound on allowable stack size for the program; the maximum allowable size is used as m when checking for positive cycles in the graph. This check can be performed in time $O(V \cdot m)$, which is linear in V when m is constant.

A graph with no negative edges and no positive cycles cannot contain any cycles except those that are zero-weight cycles. Zero-weight cycles without negative-weighted edges can only be null cycles. Null cycles cannot contribute to the longest path, and thus can be *collapsed* into a single vertex without changing the cost of the longest path. Null cycles can be detected in a graph with no negative edges and no positive cycles in at worst $O(V^2)$ time [50].

A digraph with no cycles is a directed, acyclic graph, or *DAG*. For DAG's, the longest path problem can be solved in linear time, $O(V)$ [50].

4 Coloring Control Flow Graphs

Deadline analysis of the control flow graph for a program entails categorizing vertices according to their proximity to the deadline. In broad terms, these categories can be thought of as "close", "far", and various shades of "don't know". This section presents a realistic example fragment of interrupt-driven code, and demonstrates how the corresponding CFG can be *colored* to assign each vertex into one of the categories.

Deadline analysis begins with stack size analysis. This section presents a running example to illustrate the analysis (Section 4.1), constructs an initial flow graph (Section 4.1.1), and finds the bound for the program's stack (Section 4.1.2). Next, the graph is reweighted with execution times on the edges instead of stack deltas, and graph coloring begins (Section 4.2). An iterative process of adding time summary oracles (Section 5) and recoloring the graph with additional resolution (Section 6) continues until the analysis is complete.

4.1 Example

The example program shown in Figure 2 is a short excerpt of Z86 assembly code designed to exhibit interrupt latency characteristics hostile to static analysis. As a result, the corresponding flow graph in Figure 3 is larger than one would expect for a code segment of this size. There are two vectored interrupt handlers, `IRQVC0` and `IRQVC1`, both of which do

```
    .ORG  %00h         ;INTERRUPT VECTOR TABLE
    .WORD #IRQVC0      ; Vector IRQ0
    .WORD #IRQVC1      ; Vector IRQ1
    .ORG  %0Ch
  INIT:                ;INITIALIZATION
0C   CALL PROC         ; Call a little procedure.
0F   CALL PROC         ; Call it a second time to introduce
                       ;    an artificial yellow cycle.
12   LD   IMR,  #81h   ; Enable global interrupts and IRQ handler 0.
  OUTLOOP:             ;OUTPUT LOOP
15   LD   P3,   r1     ; Send the contents of r1 out data port 3.
17   DJNZ r1,   OUTLOOP ; Dec r1, jump to top of loop if not zero.
19   CLR  IMR          ; Disable interrupts.
  BSYLOOP:             ;INPUT LOOP
1B   TM   P2,   #80h   ; Check the high bit on data port 2.
1E   JR   NZ,   BSYLOOP ; If the bit is 1, continue looping.
20   LD   IMR,  #83h   ; Enable global interrupt handling,
                       ;    and both handlers 0 and 1.
  LOOP:                ;MAIN PROGRAM
23   JP   LOOP         ; An infinite loop.
                       ;SUBROUTINES
  PROC:                ; This subroutine just pushes a value
26   PUSH r0           ;    onto the stack, and then pops it back
28   POP  r0           ;    off before returning.  Its sole purpose
2A   RET               ;    is to confuse the analysis tool and
                       ;    demonstrate the benefits of adaptive
                       ;    slicing.
                       ;INTERRUPT HANDLERS
  IRQVC0:              ; Both of these handlers do nothing except
2B   IRET              ;    execute the return from interrupt
  IRQVC1:              ;    instruction.  Even so, the complexity
2C   IRET              ;    that arises from having both in play
    .END               ;    at the same time causes all five colors
                       ;    from our analysis to appear.
```

Figure 2: Example Program

nothing but execute the return-from-interrupt instruction, IRET. The procedure PROC pushes a value from a register onto the stack, pops it off, and returns. The main loop, LOOP branches to itself infinitely. The OUTLP loop outputs the bytes 255 through 1 to an external data port and terminates, while the BSYLP loop waits until data from an external port arrives with 0 as the most significant bit.

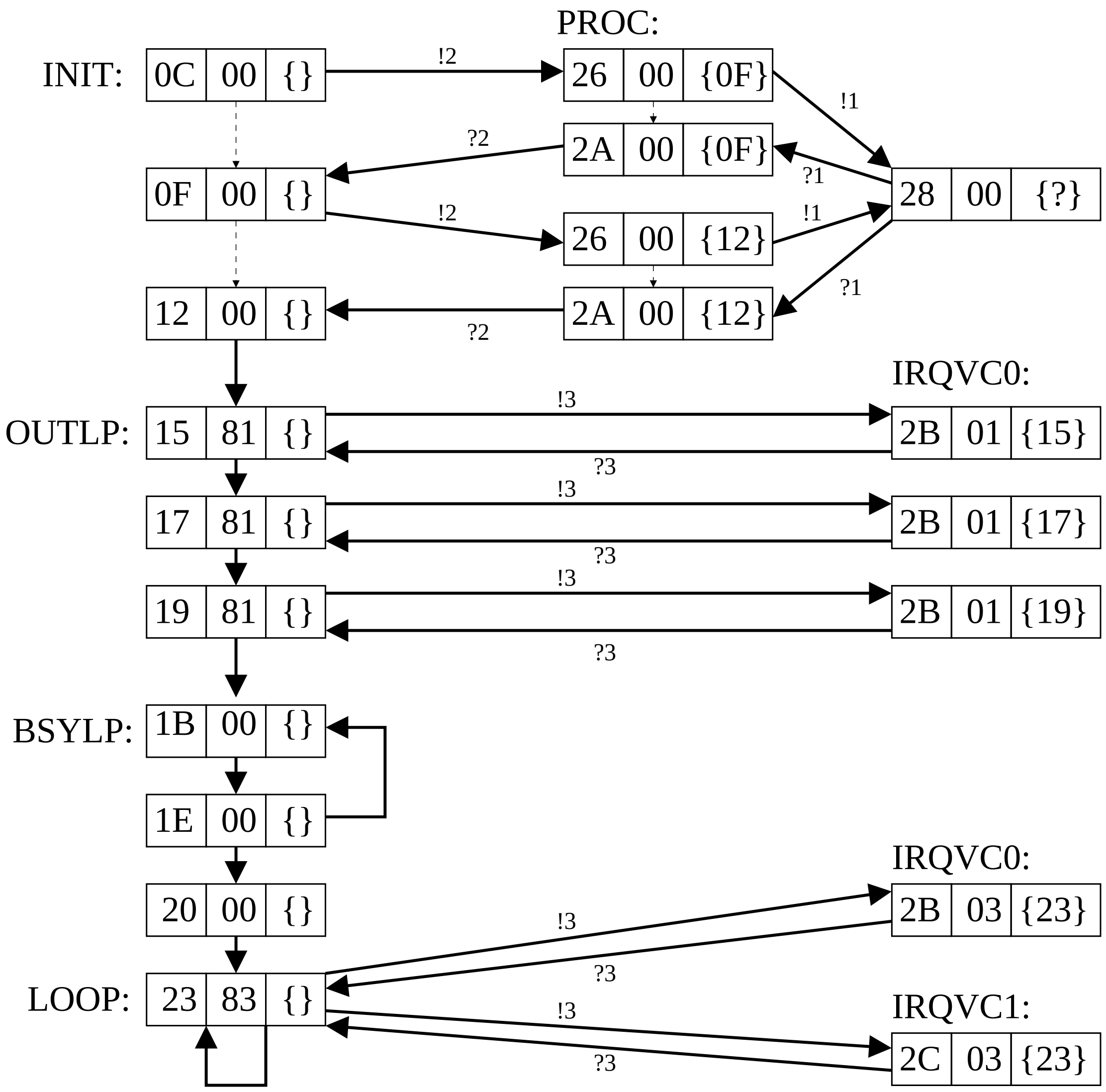


Figure 3: Example Program Flow Graph With Stack Annotation Edges

The two-digit hexadecimal numbers along the leftmost column of the figure are the ROM addresses that would be generated for this program if it were actually compiled into machine code. These addresses will be used throughout the rest of this section to refer to specific lines in the example.

4.1.1 Example Flow Graph

Figure 3 shows the flow graph constructed for the example program in Figure 2. Each vertex in the graph has three pieces of information:

- Code address – the value of the instruction pointer when the processor begins executing

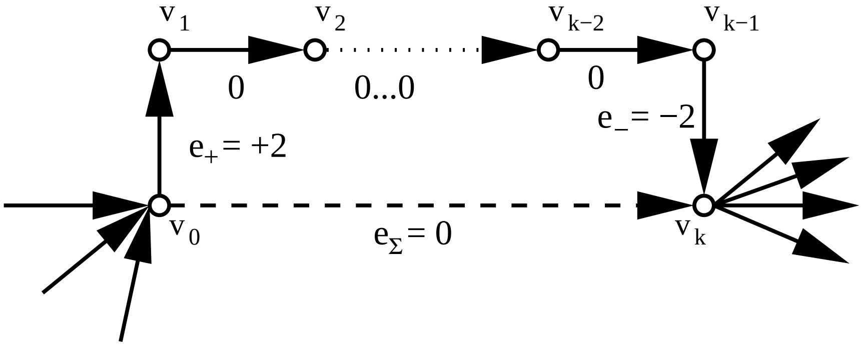


Figure 4: Summary Edge Closure in Stack Analysis Graphs

the instruction. The upper leftmost node in the graph ("INIT") contains address "0C", which is the first instruction executed by the Z86 processor on powerup.

- IMR value – the bits in the Interrupt Mask Register control vectored interrupt handling by the Z86 processor. The layout of the IMR is "M.543210", where bit "M" controls global interrupt handling, and the lower order bits enable the six correspondingly-numbered interrupt sources. The seventh bit is reserved. The node at INIT has IMR value "00", indicating that all interrupts are turned off, while the node at LOOP has IMR value "83", indicating that vectored interrupt handling is turned on and the handlers for interrupts 1 and 0 are enabled.

- Call string suffix – initially, this field contains the top element on the system stack, "{}" for an empty stack, or "?" when the exact value on the top of the stack is irrelevant. As shown later, multi-resolution analysis may add additional items of stack context to the call string suffix of a vertex as needed.

Solid edges in the graph represent possible control flow between vertices. When the transition between two vertices involves a change in the stack, the edges have been annotated with "!" and "?". The notation "!3" indicates an operation that pushes three bytes onto the stack – an interrupt. (When an interrupt handler is invoked, the Z86 pushes two bytes of return address and one byte of condition code bits onto the stack.) The notation "?2" indicates two bytes being popped off of the stack – a return from a procedure call. Dashed edges in the graph represent stack summary edges, as defined in the next subsection.

4.1.2 Stack Size Analysis

Prior to coloring a CFG for deadline analysis, it must be known that the program represented by the CFG has bounded stack size. The methods we employ for checking this property are discussed extensively in our earlier paper with Damgaard on static checking of interrupt-driven software [9].

Stack size analysis begins with a CFG in which the edge weight function equals the change in stack height seen by the program when moving between the source state and destination state. Such labels can be seen in the example graph of Figure 3. For the Z86 architecture, the edge weights can range over $[-3, 3]$.

A *summary edge* e_Σ in the graph has weight zero, a source vertex v_0, and a destination vertex v_k such that there exists a path π_Σ from v_0 to v_k in which the edge $e_+ = (v_0, v_1)$ has a positive weight, edge $e_- = (v_{k-1}, v_k)$ has an equal but opposite negative weight, and the subpath from v_1 to v_{k-1} is a null path. An example summary edge is shown in Figure 4. The first and last edges in π_Σ are said to be *matched*, since they have the same absolute value of weight, with opposite polarity. Because π_Σ consists of two matched edges and a null path, the total cost of π_Σ is zero.

A graph is said to be *closed* with respect to summary edges if and only if every non-zero-weighted edge is part of a zero-weighted path π_Σ, and thus associated with a summary edge e_Σ. Closed graphs cannot contain a negative edge e_- that does not have a matching e_+. Likewise, a closed CFG cannot contain an e_+ that does not have a matching e_-, or a summary edge e_Σ with a non-zero weight. These conditions correspond to the type-checking of stack elements to ensure that pushes match pops, procedure calls match returns, etc.

Summary edges summarize well-structured zero-weight paths in such a way that all negative-weighted edges can be deleted from the graph without altering the length of the longest paths. In a summary edge closed graph, any path from *terminus* through a negative-weighted edge must pass through an equal and opposite positive-weighted edge. If a longest path passes through a negative-weighted edge, then there exists another path of equal length passing through the associated summary edge instead. If a longest path does not pass through a negative-weighted edge, then again no negative-weighted edges were required. Summary edge closure is a key property that allows all negative-weighted edges to be removed from the graph without altering the length of any longest paths.

A graph with no negative cycles can be closed with respect to summary edges in time polynomial in V [32]. Termination of the closure algorithm guarantees that the program has a known, bounded stack height. This bounded stack height is a requirement for the deadline analysis algorithm to terminate.

4.1.3 k-CFA

At this stage of the analysis, the control flow analysis is 1-CFA [52], meaning that each vertex contains a call string suffix of at most one call site. This is relatively imprecise and inexpensive as control flow analyses normally go, but has proven sufficient to bound stack height in a solid majority of the interrupt-driven software we have examined. A 0-CFA analysis (one in which no stack context is stored at the vertices) is not sufficiently precise even for stack height analysis in realistic programs. Later sections describe how deadline analysis requires some portions of the graph to store additional stack context, k-CFA for values of k greater than 1. This more costly control flow analysis incurs additional cost both in analysis time and in size of the vertex state space.

4.2 Initial Coloring of the Example Graph

The designer of the example program in Figure 2 would like to know if the tasks corresponding to interrupts 0 and 1 will meet their deadlines. This requires information about the minimum inter-arrival time for each interrupt source. But even before that kind of data can be considered, there is another key piece of information that any such analysis must have: the WCET of the program with respect to interrupt latency. The maximum possible delay between the arrival of an interrupt request and subsequent handling of that request must be known in order to make any accurate statement about the system's ability to meet deadlines.

In order to perform deadline analysis for a given interrupt, we first annotate the CFG with a new edge weight function – each edge cost now reflects the number of machine cycles taken by the program to transition from one program point to another. Calculating the worst-case interrupt latency is now another instance of the longest path problem on the new CFG. However, the longest path cannot yet be calculated because the CFG may contain many cycles which cannot be eliminated without additional information about the program.

We classify the vertices in the flow graph into five colors. Three of those colors will be explained here; two more will be covered in Section 6.

- *Green* nodes in the graph are those from which computation will inevitably reach the handler of interest. For a green node, the analysis can compute the WCET from the node to the handler in linear time (see Section 5.1).

- *Red* nodes are those from which it is impossible to reach the handler of interest. In our model of computation, this would be a significant program error, such as an infinite loop with interrupt handling disabled. The test suite of production microcontroller software contained no such errors, so red will not be discussed any further.

- *Yellow* nodes are those which could not be definitively classified as green or red for the handler of interest.

When the analysis colors the example system flow graph (Figure 3) with respect to interrupt handler 1, the nodes with addresses `2C`, `23`, and `20` are colored green, as is the node for the lowest instance of the interrupt zero handler, `2B`, off of the `LOOP` node. Nodes `1B` and `1E` are colored yellow because the analysis cannot statically determine how long it will take to complete the `BSYLP` loop. Finally, since the remaining nodes in the graph above `BSYLP` can reach interrupt handler 1 only through `BSYLP`, they too will be colored yellow in the initial round.

Eliminating all yellow nodes in the graph would allow the analysis to give firm bounds on the execution time of any path in the program leading to the interrupt handler. The yellow nodes fall into five basic categories:

- *External Yellow* nodes comprise a cycle that depends on external input. These cannot be resolved through static analysis, and will require some form of additional information about the external environment of the controller. (For example, the node with PC value `1B` in Figure 3 is part of an external yellow cycle.)

- *Ultra Yellow* nodes comprise a cycle in the graph corresponding to some kind of unbounded loop.

- *Starvation Yellow* nodes are yellow because the interrupt handler of interest can be *starved* (delayed indefinitely [11]) by another interrupt source calling its own handler frequently enough to prevent the processor from making progress toward the handler of interest. (Nodes `15`, `17`, and `19` in the example can be starved by the handler starting at `2B`.)
- *Artificial Yellow* nodes comprise unrealizable cycles that appear in the graph as a result of implicit path merging. (The cycle of `0F`, `26`, `28`, and `2A` in the example is an artificial yellow cycle.)
- *Upstream Yellow* nodes are yellow only because they are upstream of other yellow nodes. (Nodes `0C` and `12` in the example are upstream yellow.)

Intuitively, yellow represents a "don't know" category of nodes which lie along positive cycles in the CFG. External and ultra yellow nodes can be dealt with through the use of oracles, as explained in the next section. Artificial yellow nodes are eliminated using adaptive slicing, as outlined in the section on multi-resolution analysis. Starvation yellow nodes will be assigned a new color, to be dealt with by simulation and testing. Finally, upstream yellow nodes will disappear when the other four classes of yellow nodes are eliminated.

5 Testing Oracles

Real-time, interrupt-driven software can contain loops that cannot be bounded through static analysis. Synchronous communication with off-chip resources, decisions predicated on external data, or interaction with the user can be expressed as loops whose bounds depend on additional information outside the realm of the system source code.

The `BSYLP` area of the example system is such a loop. It is a simplified version of a busy-wait loop found in several of the production microcontroller systems we have examined. Typically, such a loop could be waiting for a peripheral device to signal that it has received the last command, and can be issued further commands. The designers of the system would know that the manufacturer of the device guarantees the maximum response time for this operation will be, for example, 40ms, a fact that cannot be ascertained from the source code. In order to take advantage of this external information the analysis uses an *oracle*, an entity that answers questions about latency that cannot be answered by static analysis.

An oracle gives an assertion of the form:

$$Address_1 \rightarrow Address_2 = Latency$$

which says that the program will take at most *Latency* machine cycles to get from $Address_1$ to $Address_2$.

When constructing the initial control flow graph, information provided by the oracle is used to insert *time summary edges* from a node N in the graph with address $Address_1$ to a node M in the graph with address $Address_2$ such that M and N have the same IMR value and stack context. It was initially anticipated that the analysis would need more complex syntax for specifying oracle edges, such as pattern matching on IMR values or

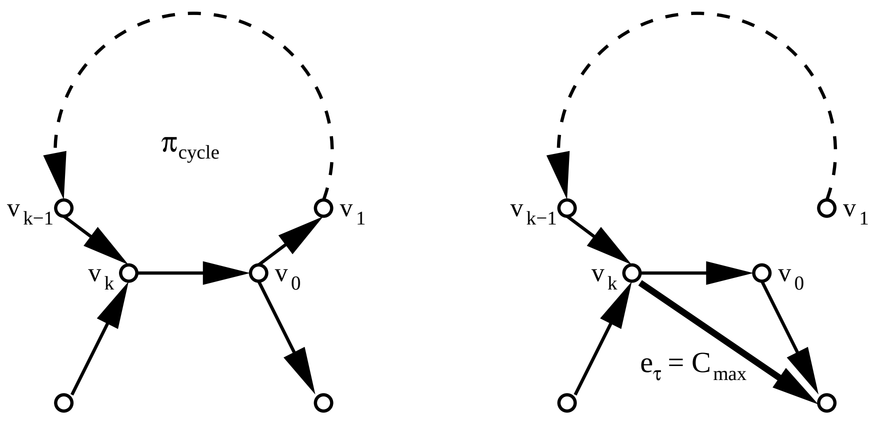


Figure 5: Time Summary Edge

stack arithmetic. However, in the six production microcontroller systems examined, the address-matching-only edges have proven sufficient to bound all of the external yellow loops.

The semantics of these time summary edges is such that the color of the destination node can be safely extended backward to the source node of the summary edge. This does *not* in itself imply anything about maximum latency between nodes that lie along a path from the source to the destination. The time summary applies strictly to the maximum latency between two nodes touched by the time summary edge.

5.1 Time Summary Edges

At this stage of the analysis, the CFG has an edge weight function w that associates each edge in the graph with a positive integer execution time count. The final WCET to the interrupt handler will be calculated as a *multi-source* longest path problem [50], which can be computed in linear time once the graph has been transformed into an acyclic digraph.

Given a weight function ranging over positive integers, there can be no negative cycles in the graph. However, positive cycles are common in deadline analysis CFG's because they correspond to the iterative control flow produced by looping constructs. Positive cycles must be removed from the graphs before deadline analysis can take place, because the algorithm does not consider the longest path to be defined in CFG's with positive cycles.

Given a positive cycle π_{cycle} and a maximum cost bound C_{max} that has been asserted by the oracle to be the maximum cost of any path along π_{cycle}, the cycle can be replaced with a *time summary edge* of weight C_{max} as shown in Figure 5.

The validity of assertions made by the user to the oracle are taken for granted by the current system. In order for the deadline analysis algorithm to remain conservative, time summary edges must be conservative. That is, a time summary edge can overestimate the true execution time of the loop it summarizes, but it cannot underestimate. If underestimated time summary edges exist in a graph, the deadline analysis algorithm is not guaranteed to

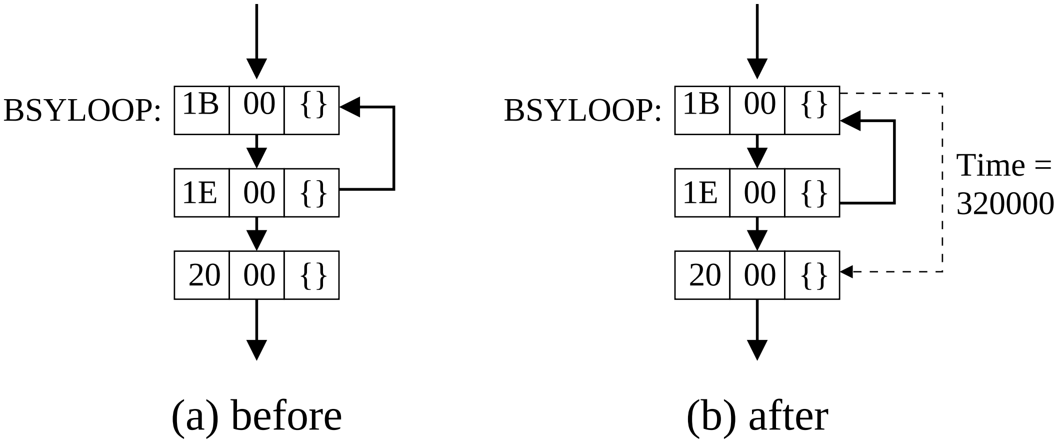


Figure 6: Time Summary Oracle in the Example

arrive at correct bounds.

Time summary edges cannot be used to summarize cycles in all cases; there exist ill-formed loops for which no single time summary edge is sufficient. Section 8.2 presents results of a practical study of time summary edges required for real programs.

In practice, one would want to concentrate system testing or simulation on areas with time summary edges to gain confidence in the validity of the assertions. However, the key point to be made is that the static analysis has greatly reduced the sheer volume of program states that must be tested. In each of the production microcontrollers analyzed, there were fewer than 20 overall assertions to the oracle, each of which covered only a handful of nodes in the graph, out of tens or hundreds of thousands of nodes in the graph overall.

Static analysis can reduce the size of the latency testing problem from an utterly intractable scale down to a subset of the program small enough that one could conceivably use exhaustive simulation to ascertain the remaining WCET information, or apply other finer-grained and less-scalable analyses.

5.2 Time Summary Edges in the Example

For the example program, a time summary oracle specifies that the `BSYLP` loop takes at most 320,000 machine cycles (40ms on the example architecture). The input to the oracle is:

```
[0x001B] -> [0x0020] = 320000
```

The resulting change to the graph is shown in Figure 6. The time summary edge from `1B` to `20` (which is already a green node) allows `1B` to be recolored green. This in turn causes `1E` to be recolored green as well, so this oracle edge has eliminated `BSYLP` as an obstacle to determining maximum interrupt latency for the entire program.

We use oracles in three ways:

- *External* event delays – bounds for loops that rely on data external to the system, such as bytes arriving on the input ports of the processor.
- *Internal* loop bounds – many of the for-loop style constructs could be bounded using well-known static analysis techniques [36, 18]. However, implementing the proper structural loop analysis for assembly language source, without any annotations from the programmer, could be far more expensive than ascertaining the loop bounds manually. Many of the loops found in the benchmarks are trivially bounded by casual examination of the code, and the time summary oracle construct is sufficiently general to bound the maximum loop execution time. This would *not* be a preferred use of the tool in practice. An industrial strength version of ZARBI would infer these bounds statically, or interactively assist the programmer in annotating the code with proper bounds. The current tool leaves this for future work.
- Internal *data* dependent loop bounds – a small number of loops in the test suite relied not on immediate constants near the top of the loop, but rather on data elsewhere in the program. The most common example of this was a display routine that iterated over a zero-terminated ASCII string. Techniques exist to automatically infer these kinds of bounds, but for simplicity of implementation, these were not employed. Instead, bounds on these loops were manually ascertained, and equivalent time summary edges were inserted.

Fully two thirds of the input provided to the time summary oracle for these experiments were loop bounds that could either be statically checked as annotations or statically inferred by other means. The remaining third of the input was for external event delays of the kind that could not possibly be determined statically. A very small number of the input items were for loops dependent on internal data, which could probably be determined with a very thorough flow analysis of all registers in the program.

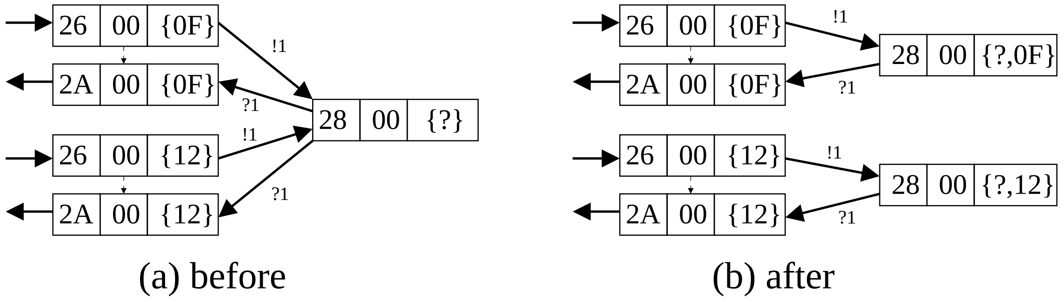
The interface provided to assist the user in giving these assertions to the oracle is quite straightforward. After initial coloring of the graph, the tool produces a list of *border yellow* nodes – yellow nodes that are one edge away from green nodes. Typically, these will be branch or jump instructions that comprise the bottom of a loop. In the case of the example program, the prototype tool would produce the result,

```
Border Yellow instructions:
        L001E:  JR      NZ,     L001B
```

directing the user to the `BSYLP` loop.

6 Multi-Resolution Analysis

Initial construction of the control flow graph includes estimates of the possible IMR values and top stack elements for each node. Abstracting away the rest of the machine state implicitly merges control flow paths, thereby allowing the size of the graph to remain tractable – typically much less than a million nodes, rather than the 2^{27} nodes which is the worst case for this model. (7 bits of IMR, 10 bits of stack element, and 10 bits of PC = 27 bits per



(a) before

(b) after

Figure 7: Example Program Adaptive Slicing

node.) However, the imprecision of having nodes distinguished by only one element of stack context (analogous to 1-CFA in flow analysis parlance [52]), can result in artificial cycles appearing in the control flow graph.

Such is the case in the example program, where procedure `PROC` is called twice within a segment where interrupt handling is disabled. Ignoring for a moment the question of how to bound latency from node `12`, the `INIT` segment of the graph would still be colored yellow because of the path [0F,00,{}], [26,00,{12}], [28,00,{?}], [2A,00,{0F}], and back to [0F,00,{}]. This is a false path [4], which does not correspond to realizable control flow – the second call to `PROC` will return to the originating call site, not the previous call site.

The approach to multi-resolution analysis shown here improves the control flow graph by eliminating many unrealizable paths.

False paths are a well known problem in control flow analysis; one solution is to employ k-CFA with larger values of k. However, it could be expensive to recompute the entire control flow graph with a higher value of k, as this quickly causes a combinatorial explosion in graph size for interrupt-driven software. Instead, the CFG is constructed using *multi-resolution analysis*, where the value of k (the size of the call string suffixes) is increased only in the areas of the graph where it is necessary to alleviate ambiguity in latency analysis. Thus, nodes like [28,00,{?}] in the example are adaptively sliced into non-yellow nodes with greater stack context, [28,00,{?,0F}] and [28,00,{?,12}], as shown in Figure 7. This approach is inspired by Plevyak and Chien [43]. Independently of our work, Guyer and Lin [25] have also used multi-resolution analysis.

Multi-resolution analysis takes place automatically; the algorithm iteratively identifies nodes that are both *border yellow* and stack popping instructions (`POP`, `RET`, and `IRET`), and adaptively slices these nodes and their associated graph segments to the necessary depth. This technique represents a substantial savings in graph complexity, reducing the size of the graph by 20% to 60% compared to running the analysis of the production programs with a fixed, non-adaptive k-CFA. However, the reduction in graph size can come at the cost of increased analysis time, as mentioned below.

While the multi-resolution analysis reduces the number of nodes and edges in the graphs in all cases, when compared with the running time of straight k-CFA, it runs faster in some cases, but slower in others. In two cases, the multi-resolution analysis is an order of magnitude slower than straight k-CFA. This wide variation in relative run times is highly dependent on the structure of the program under analysis – the depth that the adaptive slicing must go to in order to disambiguate latency, the number of call sites involved, and the lengths of the subroutines being sliced are all factors in the cost of multi-resolution analysis. For this reason, the prototype tool includes a command-line option which tells it to use straight k-CFA with a specific k, rather than automatic multi-resolution analysis, so that the user can choose whichever method performs better for their given program input.

The multi-resolution analysis is guaranteed to terminate because the control flow graphs have a bounded stack size, as verified by a previous phase of the tool. The full details of the adaptive slicing can be found in [10].

Time summary oracles allow the deadline analysis to resolve both external and internal yellow loops. Multi-resolution analysis slices apart artificial yellow nodes. Of the five types of yellow nodes, all that remain are starvation yellow and upstream yellow.

Because these nodes are yellow for a fundamentally different reason than the other nodes dealt with thus far, a new color is designated for them.

- *Magenta* nodes are those which are one edge away from either green or magenta nodes in the graph, AND are one edge away from a non-green interrupt handler.

Magenta nodes are set aside as a special case for which maximum latency of the green interrupt handler cannot be bounded without additional, detailed meta-knowledge about the characteristics of the other non-green interrupt handlers involved, (such as inter-arrival times of interrupts, jitter, etc). These nodes are also different in that the straightforward oracle-inserted time summary edges cannot help render these nodes green, even if the oracle provides bounds on the WCET of the segment of magenta nodes. This is because each magenta node can be starved, since the non-green interrupt handler can in the worst case execute so frequently that the computation does not make progress from the magenta node. (This is a point on which the Z86E30 documentation is vague; it is not clear whether an interrupt can occur frequently enough to completely halt progress in the non-interrupt code. In the absence of a clear answer, the worst case is assumed. But whatever the processor-specific behavior, either no progress or very little progress will be made in the main program under the worst-case interrupt arrival conditions, and starvation will be severe in either case.)

The WCET of contiguous clusters, or *clouds*, of magenta nodes cannot be reasoned about at the individual node level, unlike all of the other analyses presented here. For this reason, the problem of bounding *magenta clouds* is left as future work beyond the scope of this paper. Fortunately, the current analysis has revealed that on average, fewer than 2% of the nodes in the production microcontroller suite are magenta; in several cases, there are no magenta nodes at all.

Those yellow nodes which are upstream of the newly designated magenta nodes are also assigned a new color.

- *Blue* nodes are those for which the deadline analysis algorithm can precisely bound the WCET to reach a cloud of magenta nodes.

Color		**CTL formula**		**Intuition**
$\mathit{UltraGreen}$	$\equiv$	$H \vee EX(H)$	:	Head of handler of interest or one edge away.
Green	$\equiv$	$AF(\mathit{UltraGreen})$	:	Inevitable that computation will reach an UltraGreen node.
Red	$\equiv$	$\neg EF(\mathit{UltraGreen})$	:	Not possible to reach an UltraGreen node.
$\mathit{Magenta}$	$\equiv$	$EF(\mathit{UltraGreen}) \wedge EX(\mathit{handler})$ $\wedge\ AX(\mathit{handler} \Rightarrow \neg(\mathit{Green} \vee \mathit{Red}))$	:	Can reach both a Green and a non-Green handler
Blue	$\equiv$	$EF(\mathit{Magenta}) \wedge \neg EX(\mathit{handler})$ $\wedge\ AF(\mathit{Magenta} \vee \mathit{Green})$	:	Inevitable that computation will reach a Magenta node.
Yellow	$\equiv$	$\neg(\mathit{Green} \vee \mathit{Red} \vee \mathit{Magenta} \vee \mathit{Blue})$	:	Don't Know.

Figure 8: Coloring of a Graph for Latency Analysis

Intuitively, blue nodes are well-behaved segments of the graph which would be green if there were not a magenta cloud of potential interrupt starvation between them and the green handler, as suggested by Figure 1.

7 Implementation Issues

We now present details of our implementation of graph coloring and adaptive slicing. The implementation does those tasks in an integrated fashion.

7.1 Coloring Algorithm

The algorithm for coloring a graph is presented in Computation Tree Logic notation [17] in Figure 8. H is a predicate that is true for a node which represents the first instruction of the interrupt handler of interest. Predicate *handler* is true for a node which represents the first instruction of *any* interrupt handler (so, $H \Rightarrow \mathit{handler}$). We use CTL notation to talk about edges and paths to a node that satisfy a condition p:

$EX(p)$ means there is an edge from the current node to a node that satisfies p.
$AX(p)$ means every edge from the current node is to a node that satisfies p.
$EF(p)$ means there is a path from the current node to a node that satisfies p.
$AF(p)$ means every path from the current node reaches a node that satisfies p.

A node is *UltraGreen* if it either satisfies H or there is an edge to a node that satisfies H (as expressed with $EX(H)$). In other words, handling of the interrupt has just started or will start in the next computation step. The definition $\mathit{Green} \equiv AF(\mathit{UltraGreen})$ means that *Green* nodes are those for which all outgoing paths inevitably reach *UltraGreen* nodes. Thus, for *UltraGreen* nodes we can compute the WCET from the node to the handler. Notice that all *UltraGreen* nodes are also *Green*. A *Red* node cannot reach the handler of interest (as expressed with $\neg EF(\mathit{UltraGreen})$). A $Magenta$ node can reach the handler of interest (as expressed with $EF(\mathit{UltraGreen})$), but it also has an edge to some other handler (as expressed with $EX(\mathit{handler})$) which is neither *Green* nor *Red* (as expressed with $AX(\mathit{handler} \Rightarrow \neg(\mathit{Green} \vee \mathit{Red}))$). This other handler may be able to prevent the current node from reaching any *UltraGreen* nodes if the interfering interrupt or interrupts occur repeatedly. A *Blue* node can reach a $Magenta$ node (as expressed with $EF(Magenta)$) but it cannot reach any handler in one step (as expressed with $\neg(EX(\mathit{handler}))$, and every path will lead to either and $Magenta$ node or a *Green* node (as expressed with $AF(Magenta \vee \mathit{Green})$). In other words, a *Blue* node is upstream from $Magenta$ nodes and we can compute the WCET from the node to the $Magenta$ nodes. It is straightforward to show that every node has exactly one of the colors *Green*, *Red*, $Magenta$, *Blue*, and $Yellow$. The ZARBI implementation of the coloring algorithm is detailed in [10].

Let us consider the insertion of a time summary edge $A \rightarrow B = \mathit{bound}$, where A, B are nodes in the graph. We require that (1) A can reach B, (2) A is $Yellow$ or *Blue*, and (3) B is *Green*. In other words, the time summary edge is a shortcut from A to B which creates a one-step link from a $Yellow$ or *Blue* node to a *Green* node. The insertion of $A \rightarrow B = \mathit{bound}$ is done by inserting an edge from A to B and removing all other edges from A. It is straightforward to show that if we recalculate colors after inserting such time summary edges, the only possible color changes are that $Yellow$, $Magenta$, and *Blue* nodes may have become *Green*. The insertion procedure can easily be extended to allow the simultaneous insertion of multiple time summary edges.

Returning to the control flow graph from Figure 3, the three nodes at `15`, `17`, and `19` are colored magenta. The interrupt handler nodes, `2B`, hanging off of the magenta section are considered blue. The entire segment above `OUTLP`, with the help of the slicing explained in the previous section, is colored blue.

All edges in the CFG are annotated with execution cycles; all timing information is taken from the Z86 reference manual [59]. The entire flow graph of the example program is now green, blue, or magenta. The magenta cycles cannot be statically bounded, but the green and blue nodes can be broken into directed, acyclic subgraphs, each of which can be evaluated for WCET by a recursive traversal in which

$$WCET(B) = max(WCET(A) + edge_{AB})$$

where A ranges over all nodes that connect directly to node B, and $edge_{AB}$ is the cost of the edge from A to B. Running this traversal over the green nodes in the example program produces a WCET time of 320010 machine cycles between the magenta node at `19` and the interrupt handler at `2C`. The same calculation over the blue subgraph reveals a maximum WCET of 102 machine cycles from the start of the program to the start of the magenta nodes.

Combining this information with additional knowledge about the magenta section, such as, it will take at most 200 cycles to get from `12` to `1B` through the magenta section, bounds the maximum interrupt latency to be 320312 cycles.

7.2 Adaptive Slicing

Our deadline analysis includes *adaptive slicing*, an automated technique for increasing the resolution of the analysis in areas of the graph where abstraction causes ambiguity. An example is presented in Section 6; the details of the implementation are presented here, with pseudocode shown in Figure 9.

In the overall scheme of deadline analysis, multiresolution analysis takes place after the initial coloring of the graph with respect to a given handler. The first pass scans backward from the ultragreen handler to collect a list of all yellow nodes which are one edge away from green or ultragreen nodes. These nodes are the *border yellow* nodes and are the primary candidates for both adaptive slicing and time summary oracle assertions.

```
repeat
  $BorderYellowSet \Leftarrow getBorderYellow()$
  for all $borderYellow$ such that $borderYellow \in BorderYellowSet$ do
    if $borderYellow$ is a pop node then
      if $\exists edge \in \Omega(borderYellow)$ such that $destination(edge) \in Yellow$ then
        if $destination(e)$ not in non-green handler then
          for all $node \in destination(\Omega(borderYellow))))$ do
            $maxOutgoingK \Leftarrow max(contextAt(node), maxOutgoingK)$
          end for
          if $maxOutgoingK + 1 > contextAt(borderYellow)$ then
            $deletionList \Leftarrow$ all nodes backward reachable from $borderYellow$ without traversing push edges
            $rebuildList \Leftarrow$ all nodes one push edge back from $deletionList$
            delete the $deletionList$
            $buildContext \Leftarrow maxOutgoingK + 1$
            rebuild graph segment with $workList \Leftarrow rebuildList$
          end if
        end if
      end if
    end if
  end for
until No changes have been made in $G$
```

Figure 9: Adaptive Slicing Algorithm

Not all border yellow nodes can be recolored green through adaptive slicing or time summary oracles – some could be yellow because of loops and path mergings elsewhere in the graph. These are *upstream yellow* nodes, because their yellow classification depends entirely upon structures elsewhere in the graph. However, regardless of what percentage of the border yellow nodes is upstream yellow, it is still the case that some number of border yellow nodes *can* be recolored green with the help of slicing or oracles.

The multiresolution analysis next iterates through the list of border yellow nodes and discards any nodes which are not pop nodes. Pop nodes correspond to one of three opcodes in the Z86 assembly language – POP, RET, and IRET. Only the border yellow pop nodes are of interest for adaptive slicing, because they are the merge points in a backward traversal where stack context is lost. In other words, if a green node in the program has an incoming pop edge from a yellow POP instruction, it is the implicit merging of the node for the POP instruction with another node in a different stack context which causes an artificial yellow cycle to appear in the graph.

While filtering non-pop nodes out of the list, the analysis also checks each border yellow pop node candidate to see that it has at least one outgoing yellow edge that does not lead to a non-green interrupt handler. Pop nodes that are yellow only because of outgoing edges that lead in one step to a non-green interrupt handler cannot be recolored green with additional stack context; they will be colored magenta in a later graph coloring pass.

Finally, for each remaining candidate border yellow pop node, a *maxOutgoingK* tally is made, giving the maximum stack context value of any node reachable in one outgoing edge from the candidate. If a candidate node's *maxOutgoingK* is larger than the candidate's maximum stack context minus 1, the candidate is placed on the final adaptive slicing list. This condition prevents adaptive slicing from taking place on candidates where the stack context is already at least one more than all of the outgoing edge destinations. These nodes cannot be successfully recolored through slicing, as they already have full precision with regard to the stack information available at all of their successor nodes. An important caveat is that these nodes may not be their final color just because they were filtered out in the current pass; they may still need additional stack context to be colored green, but not before some outgoing destination node is itself sliced into nodes with greater context.

In practical terms, the *maxOutgoingK* test also provides an important component to the stopping criteria for the multiresolution analysis. Without this test on candidate nodes, the analysis could loop indefinitely trying to add greater stack context to a graph segment that is yellow for some other reason.

The multiresolution analysis iterates through the final list of nodes selected for adaptive slicing. For each node in the list, two new lists are calculated: the deletion list is the transitive closure of all nodes that can be reached by backward traversal of non-push nodes; the rebuild list is the set of push nodes bordering the deletion list. Push nodes can correspond to PUSH or CALL instructions. In the case where the deletion list includes the first instruction of an interrupt handler, the push nodes can be any instruction from which that interrupt handler can be reached in one edge.

The nodes on the deletion list are deleted. The nodes in the rebuild list are used to seed the initial worklist when the graph builder is called to reconstruct the deleted graph segment. Before reconstruction, however, the stack context ceiling is set to one item higher than whatever the highest stack context number was among all the nodes in the delete list. After reconstruction, the entire graph is recolored.

The overall stopping criteria for the multiresolution analysis is expensive to calculate. Adaptive slicing on any given run can push back the green frontier to expose new border yellow pop nodes that were not candidates in the previous scan. Thus, the entire process must be repeated – the entire loop in Figure 9 – until the list of final slicing candidates is of zero size.

Program	Lines	IRQs	Purpose
CTurk	1367	2	Agricultural control
GTurk	1687	2	Agricultural control
ZTurk	1612	2	Agricultural control
DRop	1162	3	Reverse osmosis control
Rop	1172	3	Reverse osmosis control
Serial	795	3	RS-485 network relay
Micro00	84	2	Example from [9]
ICSE01	55	1	Example from [9]
FSE03	35	2	Example from Section 4.1

Figure 10: Benchmark Characteristics

The existing adaptive slicing algorithm is not optimal in that a lot of work is needlessly duplicated during the analysis. In practice, large segments of graph can be built, deleted, rebuilt with greater stack context, deleted again, and rebuilt with even more stack context. A cleverer algorithm would instead update existing nodes with greater context, rather than completely recalculating control flow each time. However, this would add substantial complexity to the implementation, as the adaptive slicer would need a different graph building engine, distinct from the primary graph builder.

The complexity of the multiresolution analysis is surprisingly large, due both to the complexity of the stopping criteria, and the complexity of completely recoloring the graph after each slicing. Knowing when to stop looking for candidates for slicing requires global knowledge of the graph, and thus cannot be inexpensively implemented in a system that focuses on per-node operations. The current prototype is designed around the assumption that most analysis will focus on a single node and its immediate neighbors, and thus collecting global information can be more expensive.

8 Experimental Results

The following sections present experiments applying the prototype implementation of our analysis to a test suite of commercially available microcontroller systems. Following these results, Section 8.4 presents a narrative of a representative session with the tool, starting from a fresh program, and iterating the deadline analysis until all nodes are either green, blue, or magenta.

8.1 Benchmark Characteristics

The benchmarks (see Figure 10) used to evaluate our deadline analysis techniques are a collection of real-time, interrupt-driven systems programmed in Z86 assembly language and lent to us for analysis by Greenhill Manufacturing, Ltd. (www.greenhillmfg.com).

These systems operate on a descendant of the Z80 processor, the Z86E30 microcontroller,

	Percentage green		
Prog	IRQ$_1$	IRQ$_2$	IRQ$_3$
CTurk	100%	5%	.
GTurk	100%	2%	.
ZTurk	100%	2%	.
DRop	99%	62%	40%
Rop	99%	66%	37%
Serial	100%	54%	49%
Micro00	56%	45%	.
ICSE01	100%	.	.
FSE03	100%	28%	.

	Percentage blue		
Prog	IRQ$_1$	IRQ$_2$	IRQ$_3$
CTurk	0%	87%	.
GTurk	0%	94%	.
ZTurk	0%	94%	.
DRop	1%	36%	58%
Rop	1%	32%	60%
Serial	0%	44%	49%
Micro00	38%	49%	.
ICSE01	0%	.	.
FSE03	0%	57%	.

	Percentage magenta		
Prog	IRQ$_1$	IRQ$_2$	IRQ$_3$
CTurk	0%	7%	.
GTurk	0%	3%	.
ZTurk	0%	3%	.
DRop	1%	1%	1%
Rop	1%	1%	2%
Serial	0%	1%	1%
Micro00	5%	5%	.
ICSE01	0%	.	.
FSE03	0%	14%	.

	Percentage yellow		
Prog	IRQ$_1$	IRQ$_2$	IRQ$_3$
CTurk	0%	0%	.
GTurk	0%	0%	.
ZTurk	0%	0%	.
DRop	0%	0%	0%
Rop	0%	0%	0%
Serial	0%	0%	0%
Micro00	0%	0%	.
ICSE01	0%	.	.
FSE03	0%	0%	.

Figure 11: Results With Completed Oracles

with 256 bytes of RAM, 4K of program ROM, and a 12MHz clock. The resources available to such a chip are moderate at best, but this is true of millions of units of similar 4-, 8- and 16-bit embedded systems sold this year. The software for these systems was written by hand, in Z86 assembly language, and varies in size from about 800 to 1600 lines of code. The prototypes for each of these systems underwent months of testing prior to actual production, but the overall properties of these systems were still poorly understood, largely due to the lack of proper analysis tools like the one we present here.

The test suite also includes the example program from Figure 2, called “FSE03”, as well as examples from [9], called “ICSE01,” and “Micro00.” The commercial program “Fan” (included in earlier papers,) has been omitted because the stack size analysis cannot currently bound its maximum stack height (due to both positive and negative cycles in the corresponding CFG); bounded stack height is a precondition to running the deadline analysis algorithm.

8.2 Measurements

The results shown in Figure 11 give the final percentages of nodes by color after completion of the deadline analysis. For clarity of presentation, interrupt sources in the tables are numbered

		Adaptive Slicing		fixed k-CFA	
Program	Max k	Nodes	Edges	Nodes	Edges
CTurk	9	35750	51329	63904	84594
GTurk	10	140817	184724	215603	272421
ZTurk	10	127892	168104	190813	241118
DRop	5	19206	25244	46246	58510
Rop	5	21837	28731	54900	69597
Serial	3	8158	10753	19352	24775
Micro00	1	339	619	339	619
ICSE01	1	46	74	46	74
FSE03	2	18	33	21	33

Figure 12: Adaptive Slicing vs. Fixed k-CFA

as "IRQ_1", "IRQ_2", and IRQ_3. This does not imply any kind of priority relationship between the various interrupt sources, nor are these the actual interrupt source numbers from the Z86 processor; they are merely organized into columns. (E.g., Cturk has interrupt handlers for Z86 IRQ3, IRQ4, and IRQ5, and these are labeled 1st, 2nd, and 3rd IRQ respectively in the table.) Note that the tool rounds percentages down in most cases, or up in the case of percentages less than 1%, so the tables in Figure 11 may not total precisely to 100%.

Yellow nodes were entirely eliminated, and the percentages of green and blue were high. The magenta present in the final graphs was uniformly low, less than 2% on average. Several of the benchmarks had 0% magenta for a given IRQ, which means the analysis can safely and completely bound interrupt latency for those particular handlers from anywhere in the program.

The ZARBI deadline analysis tool is implemented in Java, and took less than the 128 Megabytes of available RAM to complete the analysis in all cases. The running time of the tool increases as the number of oracle assertions allows the tool to slice deeper into the graphs. Run-time varied from less than 2 seconds up to an hour for the largest benchmark (with full multi-resolution analysis), with an average run-time of 15 minutes overall. The current implementation has been optimized toward rapid prototyping and easy debugging of the tool, with little regard for running time and space requirements. It is expected that an industrial-strength version of the tool could be constructed to run more efficiently.

Figure 12 shows the sizes of the graphs generated by the analysis, both with adaptive slicing, and with a fixed k-CFA, where the value for k is fixed to the depth needed by the adaptive slicing.

As mentioned earlier, employing multi-resolution analysis results in a substantial savings in graph complexity, with multi-resolution graphs 20% to 60% smaller than the equivalent fixed k-CFA graphs. While the fixed k-CFA graphs can be constructed substantially faster in some cases, the reduction in yellow nodes offered by the multi-resolution analysis is usually far more valuable. When using the tool to iteratively discover time summary assertions for reducing yellow nodes, (as demonstrated in Section 8.4,) anything that causes larger graphs potentially creates more yellow nodes, adding more data to the output of the tool, and

	Number of Summary Edges			
Program	Total	External	Internal	Data
CTurk	15	5	9	1
GTurk	17	5	11	1
ZTurk	17	5	11	1
DRop	16	6	9	1
Rop	16	6	9	1
Serial	2	1	1	0
Micro00	0	0	0	0
ICSE01	1	0	1	0
FSE03	1	1	0	0

Figure 13: Oracle Information Provided

making the entire process increasingly difficult.

Figure 13 characterizes the number and types of assertions that were provided to the time summary oracle in order to eliminate all yellow nodes in the test suite.

In all cases, there was only one contiguous magenta cloud for each program that had any magenta nodes.

8.3 Assessment

The complete elimination of yellow nodes from the control flow graphs of the commercial microcontrollers was the primary goal in the deadline analysis experiments, and this was accomplished by the algorithms presented.

The high percentage of green and blue nodes makes it possible to completely bound interrupt latency for some of the interrupt sources in some of the benchmarks, and greatly decreases the remaining work to be done in bounding the others.

The low percentage of magenta nodes in the graphs, combined with the fact that magenta nodes are constrained to a single, contiguous cloud in all of the benchmarks, paves the way for automatically bounding these most troublesome parts of the graph in the future. The only case where magenta levels reached a double digit percentage was the FSE03 example program, which was constructed to have a prominent magenta segment. In many cases, the magenta section is small enough that the total uninterrupted WCET of the magenta cloud could be less than the minimum period of the interfering interrupt handler(s), making it possible to reason about these sections with a first-order *worst-case response time analysis* [54] or by detailed simulation and testing.

The number of time summary oracle assertions necessary to eliminate yellow nodes from the benchmarks is small and manageable. Well over half of the assertions are of the type that could be automatically inferred by local data flow analysis.

The original motivating challenge for this work was to see if static analysis could significantly reduce the required testing effort. Our answer is a definitive, "yes." Our tool allows the testing problem to be reduced from needing to cover the entire state space of the system down to concentrating on a handful of key passages in the code corresponding to oracle assertions and magenta sections. It is difficult to quantify just how much of a reduction in testing we can achieve, given that bounding interrupt latency for benchmarks such as ours is a largely intractable problem without our analysis technique. With the static analysis we propose, proper testing of such systems is not only possible, it can now be undertaken with standard test coverage techniques.

8.4 User Experience

This section details the complete process of starting with a raw program, and iterating with the deadline analysis to add time summary oracle assertions until all yellow nodes are eliminated.

This example will use one of the medium sized benchmarks, `Rop`.

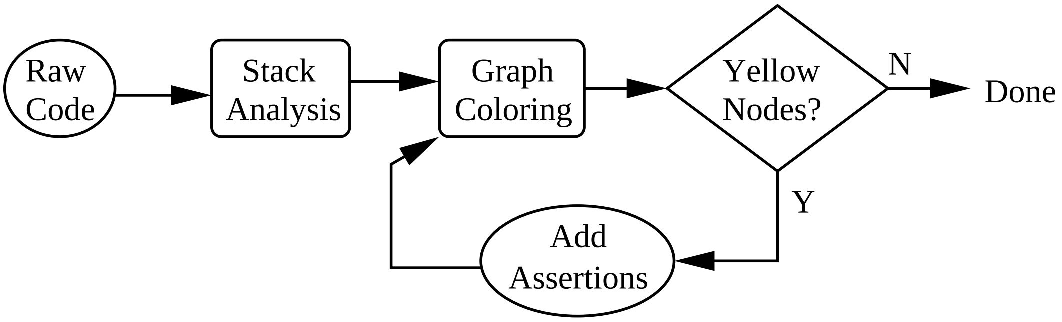


Figure 14: Tool Usage Flow Diagram

The overall process of the deadline analysis is shown in Figure 14. After an initial stack analysis, the user iterates the graph coloring deadline analysis, adding assertions until all of the yellow nodes are eliminated.

The initial run of the tool takes 23 seconds and outputs:

```
    Border Yellow instructions:
            L0667:  JR      ULT,    L0680
            L0675:  JR      ULT,    L0680
            L00D2:  JR      EQ,     L00E3
            L066C:  JR      UGT,    L067C
            L067A:  JR      ULE,    L0681
            L0312:  JR      C,      L0308
            L062D:  JR      ULE,    L061C
            L0268:  JR      UGE,    L02B7
            L0080:  JR      EQ,     L00F2
            L02BA:  JR      UGE,    L02C3
            L034C:  JR      EQ,     L0354
```

```
        L0396:  PUSH    %FBh
        L04E6:  DJNZ    r14,    L04E0

Edges = 24503   Green   Yellow  Magenta Blue
Nodes = 18559   12522   6029    2       6
Percent =       67%     32%     1%      1%
```

The list of potential yellow nodes is long for the initial run, because it is not trivial for the tool to distinguish between key yellow loops that must be broken and loop instructions that happen to be on the yellow border for other reasons.

Looking through some of the tool's suggested locations in the code, the user's attention is immediately drawn to a potential loop to bound – the `DJNZ` instruction at `L04E6` is part of a double loop that debounces the input from a mechanical switch attached to the system. The design of the system specifies that this mechanical contact should not bounce for more than 10ms when in good working order.

The double loop is actually two intertwined loops (which would be difficult to implement in most higher level languages), but can be bounded with a pair of assertions to the time summary oracle:

```
[0x04E0]->[0x04E8]=80000 ; Debounce. (10ms) [E]
[0x04DC]->[0x04E8]=80000 ; Debounce. (10ms) [E]
```

The syntax on the left describes the source and destination nodes, and the length of time to assert. To the right of the semi-colon, a comment documents the reason for the assertion, and the time translated into seconds. (80,000 machine cycles equals 10 milliseconds with an 8MHz clock.) The full grammar of the time summary oracle file format can be found in [10].

The user reruns the tool, with the new oracle assertions. After 31 seconds, the tool responds:

```
Border Yellow instructions:
        L0667:  JR      ULT,    L0680
        L0675:  JR      ULT,    L0680
        L00D2:  JR      EQ,     L00E3
        L066C:  JR      UGT,    L067C
        L067A:  JR      ULE,    L0681
        L0312:  JR      C,      L0308
        L062D:  JR      ULE,    L061C
        L0268:  JR      UGE,    L02B7
        L0080:  JR      EQ,     L00F2
        L02BA:  JR      UGE,    L02C3
        L034C:  JR      EQ,     L0354
        L0396:  PUSH    %FBh
```

```
          L04DA:  JR      NZ,     L04D6

Edges = 24513   Green   Yellow  Magenta Blue
Nodes = 18559   12528   6023    2       6
Percent =       67%     32%     1%      1%
```

Note that the node total has remained the same, but six nodes that were yellow are now green. The DJNZ instruction at L04E6 is no longer listed as a border yellow node, and a new border node is listed in its place. The tool also outputs the number of red nodes in the graph, if any, but none of these graphs contained red nodes.

The loop at L04DA is a holding pattern that waits for the human operator to release one of the push buttons. The user interface segments of this microcontroller system are only executed when the system is in a programming mode, so attention to interrupt handlers is not important here. The user assumes that no one is pushing the button, and the branch will never be taken.

The loop at L0312 waits on an external device that the microcontroller has synchronous communication with. The manufacturer guarantees a maximum 40ms delay before the device responds.

The loop at L062D has a visible bound, but calls several levels of complex subroutines. This is the sort of loop that would be extremely tedious to estimate by hand with any accuracy, but which could probably be automatically bounded by a local data flow analysis around the loop and its subroutines. For now, the user puts in an outrageous overestimate of 3 full seconds; this area should be simulated in depth in order to tighten the estimate later.

The jump instruction JR EQ, L0354 at L034C is part of a loop that writes ASCII strings to a connected LCD panel one byte at a time. The number of iterations for the loop is dependent upon the length of the string passed into the subroutine, but the system is designed to have a 16 character LCD display, and none of the zero-terminated ASCII string constants in the program are longer than 17 characters. The subroutine called from within the loop is green from some other call sites, so with some work, the user can conservatively bound the loop to be 17 characters times at most 40ms, for a total of 680ms.

The oracle is provided with the next set of assertions. The bracketed letters on the far right of the comment are personal notes about the type of assertion. An “[E]” indicates “external delay loops,” which are impossible to statically bound. An “[A]” indicates loops dependent on internal data, and the letter “[D]” indicates a more difficult class of internal data-dependent loops.

```
[0x04D6]->[0x04DC]=30        ; No button press. [E]
[0x061C]->[0x062F]=24000000  ; Punt. (3sec) [A]
[0x0308]->[0x0314]=320000    ; Display. (40ms) [E]
[0x033D]->[0x0354]=5440000   ; 17 char (680ms) [D]
```

This run takes 36 seconds, and has reduced the number of suggested border nodes to look at. The PUSH instruction continues to appear in the list only because some other yellow obstacle is preventing the slicer from identifying the correct segment to which additional stack context should be added.

```
Border Yellow instructions:
        L0396:  PUSH    %FBh
        L0608:  DJNZ    r12,    L0601
        L0650:  JR      ULE,    L063F
        L042A:  JR      Z,      L041C

Edges = 25044   Green   Yellow  Magenta Blue
Nodes = 18992   16470   2431    2       89
Percent =       86%     12%     1%      1%
```

The loop at L042A is part of another software debouncing area. The user will assume no button press.

The loop at L0650 is a twin to the loop at L062D above, so the user duplicates the assertion edge with new source and destination addresses.

The DJNZ instruction at L0608 is part of a nested loop that was designed to wait 20ms before sending more data to a peripheral chip.

More assertions are added, and the tool is rerun.

```
[0x0420]->[0x0427]=46       ; No button press. [E]
[0x0420]->[0x042C]=66       ; No button press. [E]
[0x063F]->[0x0652]=24000000 ; Punt. (3sec) [A]
[0x0601]->[0x060A]=166086   ; EEPROM write (20ms) [A]
[0x0603]->[0x060A]=166086   ; EEPROM write (20ms) [A]

Border Yellow instructions:
        L0396:  PUSH    %FBh
        L05E5:  DJNZ    r13,    L05D8
        L05F6:  DJNZ    r13,    L05EA

Edges = 25088   Green   Yellow  Magenta Blue
Nodes = 19020   17562   1367    2       89
Percent =       92%     7%      1%      1%
```

After 39 seconds of analysis, the percentage of green nodes has topped 90%, and the remaining yellow nodes are in the single digit range. The user is in the home stretch now.

Both of the suggested DJNZ instructions belong to loops with obvious bounds. While somewhat tedious, the user is able to total up the execution time of the dozen instructions in the bodies of the loops, and multiply them by the bounds.

```
[0x05EA]->[0x05F8]=144  ; RDLP1 (8*18cyc=18uS) [A]
[0x05D8]->[0x05E7]=1200 ; SENDBF (8*150c =150uS) [A]

Border Yellow instructions:
        L0396:  PUSH    %FBh
        L0490:  DJNZ    r14,    L048D

Edges = 28728   Green   Yellow  Magenta Blue
Nodes = 21837   21242   504     2       89
Percent =       97%     2%      1%      1%
```

After a 1 minute, 19 second analysis, the program has 97% green nodes.

The next border node belongs to a loop with obvious bounds calling a 40ms subroutine. There are two very similar loops with slightly different bounds on the page above `L0490`. The user adds assertions for all three.

```
[0x048D]->[0x0492]=1601000  ; DSPBCK 5x (201ms) [A]
[0x046C]->[0x0471]=1601000  ; DSPBCK 5x (201ms) [A]
[0x0445]->[0x044A]=1280800  ; DSPBCK 4x (161ms) [A]
```

The final run of the tool takes 1 minute, 26 seconds, but produces zero yellow nodes.

```
Edges = 28731   Green   Yellow  Magenta Blue
Nodes = 21837   21746   0       2       89
Percent =       99%     0%      1%      1%
```

There is still much testing to be done for this embedded system. The user has presented 16 assertions to the oracle, 10 of those based upon manual inspection of the code, rather than external design criteria. Simulation and testing of the system should aim to validate and/or tighten these unchecked assertions.

While the two magenta nodes in the system seem to be a small window of opportunity for interrupt starvation, they comprise an infinite loop with a non-green interrupt source turned on. In other words, the system turns off all other interrupts, and waits for a particular, different interrupt to occur before returning to normal operation. Thus, deadline analysis for this system and this particular interrupt handler depends ultimately upon knowing the upper bound on the time the system will have to wait for this other interrupt source to be triggered.

Overall understanding of the example system's timing behavior has increased as a result of the deadline analysis. Testing and simulation can concentrate on the lines of code for which assertions have been provided, and on the magenta nodes, both of which comprise a tiny fraction of the total state space for the code. The prototype implementation also produces flow graphs that depict the colors of code regions, or can dump the graph in a flat file format suitable for import into other visualization tools. Additional implementation details are presented in [10].

9 Conclusion

For interrupt-driven assembly code, our tool makes it significantly easier to perform deadline analysis. We use static analysis to reduce the required testing effort to concentrate on the validity of certain testing oracles. Our multi-resolution analysis allows for compact and efficient representation of timing properties while smoothly incorporating the oracles. For each of our test programs, at most 17 oracles are sufficient, and these can be added in an interactive fashion until the deadline analysis is complete. In our experience, an expert user can go from a bare program of about 1000 lines of assembly code to a completed deadline analysis in less than an hour, not counting the testing of the oracles.

While the current incarnation of the tool uses a Z86 front end, the abstractions used in the graph analysis are applicable to a wide range of other processors which use bit-maskable, vectored interrupt handling, such as the Motorola 68000 family and many RISC DSP chips.

Several valuable lessons were learned while experimenting with our tool on real code "found in the wild." First, for designers of deadline analysis tools it is absolutely crucial to find good techniques for presenting and visualizing the vast amount of data available when analyzing even moderately sized interrupt-driven systems. For interrupt-driven system designers, it is key to note that stack boundedness is a precondition for this kind of deadline analysis. We have one commercial microcontroller system for which we cannot provide a deadline analysis because its stack behavior is too byzantine for the current incarnation of the stack analyzer to handle. Finally, for low-level, interrupt-driven systems of this kind it is essential that the full interrupt capabilities of the microcontroller be used as sparingly as possible. Global interrupts should be kept disabled whenever possible, and individual interrupts should be masked off when not needed. Judicious use of interrupts greatly reduces the complexity of the analysis problem, and ultimately reduces the size of the magenta sections that require further modeling after our deadline analysis. While we believe our work has shown that existing techniques can tackle deadline analysis for well-written, medium-sized interrupt-driven systems, it is surely the case that sloppy or naive implementation of an interrupt-driven system can still render the problem into intractability.

Future work includes improvements in (1) discovery of loop variables bounds, (2) static analysis of magenta clouds, based on specifications of minimum inter-arrival times for interrupts, and (3) the interface for visualization of the graph.

Acknowledgments. We thank Mayur Naik, Krishna Nandivada, John Regehr, Michael Richmond, Ben Titzer and the anonymous reviewers for helpful comments on drafts of the FSE paper. We thank Greenhill Manufacturing, Ltd., for the use of their proprietary software as test input. We were supported by a National Science Foundation ITR Award number 0112628.

References

[1] Agere Systems, Inc. *APP550 Student Training Guide.* 1905A Kramer Lane Suite 100, Austin, Texas 78758, May 2003.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1985.

[3] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *Proceedings of SAS 96: International Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1996.

[4] Peter Altenbernd. On the false path problem in hard real-time programs. In *Proceedings of ERTS 96: Eighth EuroMicro Workshop on Real-Time Systems*, pages 102–107, June 1996. URL `http://citeseer.nj.nec.com/altenbernd96false.html`.

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. URL `http://citeseer.nj.nec.com/alur94theory.html`.

[6] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. In J. W. de Bakker, Cornelis Huizing, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer-Verlag, 1992.

[7] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *Proceedings of CC 04: Thirteenth International Conference on Compiler Construction*, volume 2985, pages 5–23. Springer-Verlag, March 2004. URL `http://www.cs.wisc.edu/wpis/abstracts/cc04.ps`.

[8] Guillem Bernat, Alan Burns, and Andy Wellings. Portable worst-case execution time analysis using Java byte code. In *Proceedings of 12th Euromicro Conference on Real-Time Systems*, pages 81–88, Stockholm, Sweden, June 2000. URL `http://www.computer.org/Proceedings/euromicro-rts/0734/0734toc.htm`.

[9] Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt driven software. In *Proceedings of ICSE 01: 23rd International Conference on Software Engineering*, pages 47–56, June 2001. URL `http://citeseer.nj.nec.com/brylow01static.htm`.

[10] Dennis W. Brylow. *Static Analysis of Interrupt-Driven Software.* PhD thesis, Purdue University, 2003.

[11] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages.* Addison Wesley, 3rd edition, 2001.

[12] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, Boca Raton, Florida, 1997.

[13] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, Cambridge, Massachusetts, January 2000.

[14] Matteo Corti, Roberto Brega, and Thomas Gross. Approximation of worst-case execution time for preemptive multitasking systems. In *Proceedings of LCTES 00: ACM*

SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems, volume 1985 of *Lecture Notes in Computer Science*, pages 178–198. Springer-Verlag, 2000. URL `http://link.springer.de/link/service/series/0558/tocs/t1985.htm`.

[15] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL 77: Fourth Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY. URL `http://www.di.ens.fr/~cousot/COUSOTpapers/POPL77.shtml`.

[16] Matthew B. Dwyer. *Data Flow Analysis For Verifying Correctness Properties of Concurrent Programs*. PhD thesis, University Massachusetts, Amherst, September 1995. URL `http://www.cis.ksu.edu/~dwyer/papers/thesis.ps.gz`.

[17] E. Allen Emerson. Temporal and modal logic. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science (vol. B): Formal Models and Semantics*, pages 995–1072. MIT Press, 1990.

[18] Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proceedings of RTSS 00: 21st IEEE Real-Time Systems Symposium*, November 2000. URL `http://citeseer.nj.nec.com/engblom00modeling.html`.

[19] Jakob Engblom, Andreas Ermedahl, Mikael Sjödin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *Software Tools for Technology Transfer*, 14, 2000. URL `http://citeseer.nj.nec.com/engblom00worstcase.html`.

[20] Jakob Engblom and Bengt Jonsson. Processor pipelines and their properties for static WCET analysis. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Proceedings of EMSOFT 02: Second International Conference on Embedded Software*, volume 2491 of *Lecture Notes in Computer Science*, pages 334–348, Grenoble, France, October 2002. Springer-Verlag. URL `http://link.springer.de/link/service/series/0558/tocs/t2491.htm`.

[21] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In T.A. Henzinger and C.M. Kirsch, editors, *Proceedings of EMSOFT 01: First International Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485, Tahoe City, California, October 2001. Springer-Verlag.

[22] Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of LCTRTS 97: ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 37–46, Las Vegas, Nevada, 1997.

[23] David Gay, Philip Levis, J. Robert von Behren, Matt Welsh, Eric A. Brewer, and David E. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of PLDI 03: ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–11. ACM Press, 2003.

[24] Jan Gustafsson, Björn Lisper, Nerina Bermudo, Christer Sandberg, and Linus Sjöberg. A prototype tool for flow analysis of C programs. In *Proceedings of WCET 02: Second IEEE International Workshop on Worst Case Execution Time Analysis*, pages 10–13, Vienna, Austria, June 2002.

[25] Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In Radhia Cousot, editor, *Proceedings of SAS 03: Tenth Annual International Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 214–236, San Diego, California, 2003. Springer-Verlag.

[26] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, and Ashish Gujarathi. Regression test selection for Java software. In *Proceedings of OOPSLA 01: 16th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 312–326. ACM Press, 2001. URL `http://doi.acm.org/10.1145/504282.504305`.

[27] Christopher A. Healy, Mikael Sjödin, Viresh Rustagi, David B. Whalley, and Robert van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, 18(2/3):129–156, 2000. URL `http://citeseer.nj.nec.com/healy00supporting.html`.

[28] Christopher A. Healy and David B. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Transactions on Software Engineering*, 28(8):763–781, August 2002.

[29] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *Proceedings of PLDI 04: International Conference on Programming Language Design and Implementation*, pages 1–13. ACM Press, 2004.

[30] ILOG. CPLEX mixed integer optimizer. URL `http://www.ilog.com/products/cplex/product/mip.cfm`.

[31] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, Massachusetts, 2nd edition, April 1999.

[32] David McAllester. On the complexity analysis of static analyses. In *Proceedings of SAS 99: International Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 312–329. Springer-Verlag, 1999. URL `http://citeseer.nj.nec.com/mcallester99complexity.html`.

[33] Patrick C. McGeer and Robert K. Brayton. *Integrating Functional and Temporal Domains in Logic Design*, volume 139 of *Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, May 1991.

[34] Tulika Mitra and Abhik Roychoudhury. A framework to model branch prediction for WCET analysis. In *Proceedings of WCET 02: Second IEEE International Workshop on Worst Case Execution Time Analysis*, pages 68–71, Vienna, Austria, June 2002.

[35] Jeffery C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of SOSP 87: 11th ACM Symposium on Operating Systems Principles*, pages 39–51, 1987. URL `http://citeseer.nj.nec.com/mogul87packet.html`.

[36] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[37] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of ICSE 99: 21st International Conference on Software Engineering*, pages 399–410, May 1999.

[38] Gleb Naumovich and Lori A. Clarke. Extending FLAVERS to check properties on infinite executions of concurrent software systems. Technical Report TR-CIS-2000-02, Polytechnic University, April 2000. URL `http://cis.poly.edu/tr/tr-cis-2000-02.htm`.

[39] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[40] Peter Notebaert. lp_solve: Mixed integer linear program solver. URL `ftp://ftp.es.ele.tue.nl/pub/lp_solve`.

[41] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proceedings of OOPSLA 91: Sixth Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–161. ACM Press, 1991. URL `http://citeseer.nj.nec.com/palsberg91objectoriented.html`.

[42] Stefan Petters and Georg Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Proceedings of RTCSA 99: Sixth International Conference on Real-Time Computing Systems and Applications*, pages 442–449, 1999. URL `http://www.computer.org/proceedings/rtcsa/0306/0306toc.htm`.

[43] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of OOPSLA 94: Ninth annual conference on Object-Oriented Programming Systems, Language, and Applications*, pages 324–340. ACM Press, 1994.

[44] Andreas Podelski. Model checking as constraint solving. In *Proceedings of SAS 00: International Static Analysis Symposium*, volume 1824 of *Lecture Notes in Computer Science*, pages 22–37. Springer-Verlag, 2000. URL `http://citeseer.nj.nec.com/podelski00model.html`.

[45] Peter Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1(2):159–176, September 1989. URL `http://www.vmars.tuwien.ac.at/`.

[46] Peter P. Puschner and Anton V. Schedl. Computing maximum task execution times - a graph-based approach. *Journal of Real-Time Systems*, 13(1):67–91, 1997. URL `http://citeseer.nj.nec.com/puschner97computing.html`.

[47] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. In *Proceedings of EMSOFT 03: Third International Conference on Embedded Software*, volume 2855 of *Lecture Notes in Computer Science*, pages 306–322. Springer-Verlag, 2003. URL `http://www.cs.utah.edu/flux/papers/emsoft03/`.

[48] Thomas Reps. Undecidability of context-sensitive data-independence analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):162–186, 2000. URL `http://doi.acm.org/10.1145/345099.345137`.

[49] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of ICSE 92: 14th International Conference on Software Engineering*, pages 105–118. ACM Press, 1992. URL `http://doi.acm.org/10.1145/143062.143100`.

[50] Robert Sedgewick. *Algorithms in C, Part 5: Graph Algorithms.* Addison-Wesley, third edition, 2001.

[51] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In Steven Muchnick and Neil Jones, editors, *Program Flow Analysis, Theory and Applications.* Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[52] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, CMU, May 1991. CMU–CS–91–145.

[53] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Journal of Real-Time Systems*, 18(2/3):157–179, 2000. URL `http://citeseer.nj.nec.com/theiling99fast.html`.

[54] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Journal of Real-Time Systems*, 6(2):133–152, March 1994. URL `http://citeseer.nj.nec.com/tindell92extendible.html`.

[55] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of OOPSLA 00: 15th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 281–293, 2000. URL `http://citeseer.nj.nec.com/tip00scalable.html`.

[56] Emilio Vivancos, Christopher Healy, Frank Mueller, and David Whalley. Parametric timing analysis. In *Proceedings of LCTES 01: ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 88–93. ACM Press, 2001.

[57] Joachim Wegener and Frank Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Journal of Real-Time Systems*, 21(3):241–268, November 2001.

[58] Zhichen Xu, Barton P. Miller, and Thomas Reps. Safety checking of machine code. In *Proceedings of PLDI 00: ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 35, pages 70–82, 2000. URL `http://citeseer.nj.nec.com/xu00safetychecking.html`.

[59] Zilog, Incorporated. *Z86E30/E31/E40 Preliminary Product Specification.* Campbell, California, 1998. URL `http://www.zilog.com/pdfs/z8otp/e303140.pdf`.

Dennis Brylow received BSc degrees in computer science and in electrical engineering from Rose-Hulman Institute of Technology; he completed his MSc and Ph.D in Computer Science at Purdue University, where he is currently a Visiting Assistant Professor of Computer Science and Electrical & Computer Engineering. Prior to earning his doctorate, he worked as a design engineer building both hardware and software for embedded and distributed control systems. His research focuses on real-time, interrupt-driven, and embedded systems with particular emphasis on building better tools for tomorrow's embedded system designers and software engineers.

Jens Palsberg is a Professor of Computer Science at UCLA. His research interests span the areas of compilers, embedded systems, programming languages, software engineering, and information security. He has authored over 70 technical papers, co-authored the book Object-Oriented Type Systems, and co-authored the 2002 revision of Appel's textbook on Modern Compiler Implementation in Java. He is an associate editor of ACM Transactions of Programming Languages and Systems, a member of the editorial board of Information and Computation, and a former member of the editorial board of IEEE Transactions on Software Engineering.