Deadline Analysis of Interrupt-driven Software

Dennis Brylow Jens Palsberg

Purdue University
{brylow,palsberg}@cs.purdue.edu

ABSTRACT

Real-time, reactive, and embedded systems are widely 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 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.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software Verification

General Terms

Algorithms, Measurement, Reliability

Keywords

Real time, multi-resolution static analysis, testing oracles

ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.

1. INTRODUCTION

Background. 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.

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 [6], bytecode verifiers [13], and model checkers [7], 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 [2, 1]. 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 [22] is not concerned with timing properties.

Our goal is to provide tool support for checking timing properties of real-time assembly code. In this work we focus 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 systems we have analyzed 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. We are seeking tool support to

answer the following question.

> **Deadline Analysis:** Will every interrupt be handled before the 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 [5], 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 [5] 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 [12]. 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 [17] 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.

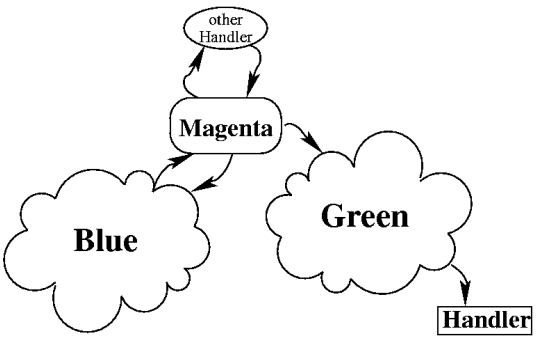


Figure 1: Coloring a Flow Graph

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 [15, 19, 9, 10, 3, 21, 8]. 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.

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

Our Results. We have designed and implemented a tool for integrated deadline and WCET analysis of interrupt-driven assembly code. In slogan form:

deadline analysis = static analysis + testing oracles.

For six commercial microcontroller programs, each on the order of 1000 lines of code, we found that less than 17 oracles were sufficient. 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 all is well, *Magenta* means that starvation is possible, *Blue* means that starvation is possible later in the computation, *Yellow* means that the analysis thinks that the deadline might not be met, and *Red* means that the analysis is certain that the deadline cannot be met. For our test suite,

```
     .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 a second time.
12   LD   IMR, #81h    ; Enable IRQ handler 0.
  OUTLP:               ;OUTPUT LOOP
15   LD   P3,  r1      ; Contents of r1 out port 3.
17   DJNZ r1,  OUTLP   ; Dec r1, jump if not zero.
19   CLR  IMR          ;Disable interrupts.
  BSYLP:               ;INPUT LOOP
1B   TM   P2,  #10h    ; Check high bit on port 2.
1E   JR   NZ,  BSYLP   ; If bit 1, busy-wait.
20   LD   IMR, #83h    ;Enable handlers 0 and 1.
  LOOP:                ;MAIN PROGRAM
23   JP   LOOP         ; An infinite loop.
  PROC:                ;SUBROUTINES
26   PUSH r0           ; Pushes value onto stack,
28   POP  r0           ; pops it back off.  Sole
2A   RET               ; purpose: confuse analysis.
                       ;INTERRUPT HANDLERS
  IRQVC0:              ; Both handlers do nothing.
2B   IRET              ; Even so, complexity that
  IRQVC1:              ; arises from both in play
2C   IRET              ; causes all five colors.
```

Figure 2: Example Program

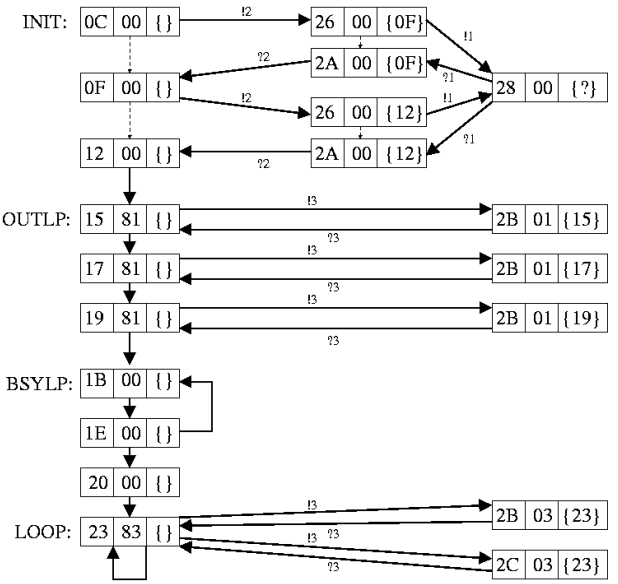


Figure 3: Example Program Flow Graph

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.

In the following section, we present a program which will be used as a running example throughout the paper. In Section 3 we present our notion of oracles, in Section 4 we show our multi-resolution static analysis, in Section 5 we discuss our experimental results, and in Section 6 we walk the reader through a session with our tool.

2. EXAMPLE

A Program and its Flow Graph. 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. There are two vectored interrupt handlers, `IRQVC0` and `IRQVC1`, both of which do 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.

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. We will use these addresses throughout the rest of this section to refer to specific lines of the example.

For the example program in Figure 2, we construct the flow graph in Figure 3. Each node in the graph has three pieces of information:

- Code address – the value of the instruction pointer when the processor begins executing 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 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.
- Stack context – 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 we will see later, multi-resolution analysis may add additional items of stack context to nodes as needed.

Solid arrows in the graph represent possible control flow between nodes. When the transition between two nodes 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 arrows in the graph represent stack summary edges, as defined in [5].

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. We must know the maximum possible delay between the arrival of an interrupt request and subsequent handling of that request 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, our tool classifies the nodes in the flow graph into five colors. Three of those colors will be explained here; two more will be covered in Section 4.

Green nodes in the graph are those from which computation will inevitably reach the handler of interest. For a green node, we can easily compute the WCET from the node to the handler.

Red nodes are those from which it is impossible to reach the handler of interest. In the model of computation we are considering, this would be a significant program error, such as an infinite loop with interrupt handling disabled. Our 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.

If we color the example system flow graph (Figure 3) with respect to interrupt handler 1, the nodes with addresses `2C`, `23`, and `20` would be colored *Green*, as would the node for the lowest instance of the interrupt zero handler, `2B`, off of the `LOOP` node. Nodes `1B` and `1E` would be colored *Yellow* because the tool 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 us 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:

- Nodes that are *Yellow* because they comprise a cycle in the graph corresponding to an unbounded loop.
- Nodes that are *Yellow* because they comprise a cycle that depends on external input. These can never 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.)
- Nodes that are *Yellow* as a result of the appearance of cycles that are artifacts of imprecision in the static analysis, such as implicit path merging. (Cycle of `0F`, `26`, `28`, and `2A` in the example.)
- Nodes that are *Yellow* because the interrupt handler of interest can be "starved" (in the classic operating system sense) by another interrupt 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.)
- Nodes that are *Yellow* only because they are "upstream" of other *Yellow* nodes. (Node `0C` or `12` in the example.)

Intuitively, *Yellow* represents a "don't know" category of nodes. The first two classes of nodes can be dealt with through the use of oracles, as explained in the next section. Infeasible-path *Yellow* nodes are eliminated using adaptive slicing, as outlined in the section on multi-resolution analysis. Starved *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.

3. 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 we have found in several production microcontroller systems. 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, e.g., 40mS, a fact that cannot be ascertained from the source code. In order to take advantage of this external information in our analysis, we use an "oracle", an entity that answers questions about latency that cannot be answered by static analysis.

Syntax and Semantics of Oracles. An oracle is 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 $Address_2$.

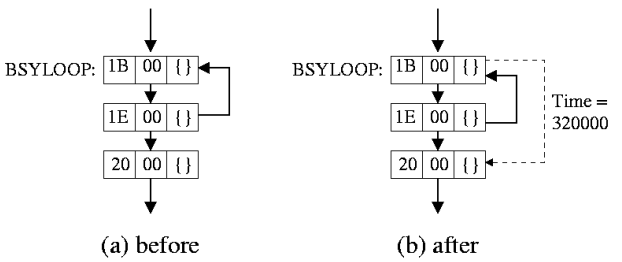
When constructing the initial control flow graph, our tool uses the information provided by the oracle to insert *time summary edges* from a node N in the graph with address $Address_1$ to a node M in the graph with address $Address_2$ such that M and N have the same IMR value and stack context. We initially anticipated needing more complex syntax for specifying oracle edges, such as pattern matching on IMR values or stack arithmetic. However, in the six production microcontroller systems we have examined, the simple address-matching-only edges have proven sufficient to bound all of the loops that depend on external data.

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.

For the example program, we use a time summary oracle to specify 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 can be shown as follows:



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.

Three Uses of Oracles. In our experiments, we have used 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 [14, 9]. However, implementing the proper structural loop analysis for assembly language source, without any annotations from the programmer, could be far more expensive than simply ascertaining the loop bounds manually (many of these loops are trivially bounded by casual examination of the code) and employing our sufficiently general time summary oracle. This would *not* be a preferred use of this tool in practice. An industrial strength version of the tool would infer these bounds statically, or interactively assist the programmer in annotating the code with proper bounds. Our current tool leaves this for future work.
- Internal *data* dependent loop bounds – a small number of loops in the production programs 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 none were already implemented for Z86 assembly language, so we chose to manually ascertain the bounds on these loops, and insert equivalent time summary edges.

Fully two thirds of the input we provided to our time summary oracle were loop bounds that could either be statically checked as annotations or statically inferred by a more advanced tool. 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 our tool provides 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, our tool would produce the result,

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

directing the user to the BSYLP loop.

The truthfulness of assertions made by the user to the oracle are taken for granted by the current system. In practice, one would want to concentrate system testing or simulation on these areas 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 we 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.

4. MULTI-RESOLUTION ANALYSIS

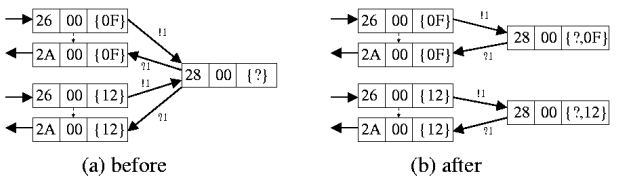
When initially constructing the control flow graph of a program, our tool uses static analysis to glean the possible IMR values and top elements of the stack for each node. Abstracting away the rest of the machine state implicitly merges control flow paths, thereby allowing the size of the graph to remain 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, 8 bits of stack element, and 12 bits of PC = 27 bits per node.) However, the imprecision of having nodes distinguished by only one element of stack context (analogous to 1-CFA in flow analysis parlance [18]), 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, which does not correspond to genuine control flow – the second call to PROC will return to the originating call site, not the previous call site.

In the following subsections we present our approach to multi-resolution analysis which improves precision of the control flow graph, thereby eliminating many false paths.

Multi-Resolution via Adaptiveness. 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 disastrous to recompute the entire control flow graph with a higher value of k, as this quickly causes the graph to explode in size for interrupt-driven software. Our tool uses a *multi-resolution analysis*, where the value of k (the amount of stack context used to distinguish nodes), 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}]:



This approach is inspired by Plevyak and Chien [16]. Independently of our work, Guyer and Lin [11] have also used multi-resolution analysis.

This multi-resolution analysis takes place automatically; the tool 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. While our 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, we have included a command-line option which tells the tool 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, which is verified by a previous phase of the tool [5]. The full details of the adaptive slicing can be found in [4].

Magenta and Blue Nodes. We have resolved unbounded loops, and external and internal data-dependent loops using oracles. We have used multi-resolution analysis to slice apart *Yellow* nodes which appear as the result of implicit path merging. We now have only interrupt starvation *Yellow* nodes to contend with.

Because these nodes are *Yellow* for a fundamentally different reason than the other nodes we have colored thus far, we designate a new color.

- *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.

We set aside these *Magenta* nodes 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 (knowledge 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 we provide the oracle with 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.

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 we have mentioned thus far. For this reason, we are forced to leave the *Magenta cloud* bounding problem to a future step of the deadline analysis process. Fortunately, our analysis has revealed that on average, fewer than 2% of the nodes in our production microcontroller suite are *Magenta*; in several cases, there are no *Magenta* nodes at all.

The previously *Yellow* nodes which were upstream from *Magenta* nodes are now also assigned a new color.

- *Blue* nodes are those for which we can precisely bound the WCET to reach a cloud of *Magenta* nodes.

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, see Figure 1.

The algorithm for coloring the graph can be summarized in CTL notation. We use H to denote a predicate that is true for a node when it represents the first instruction of the interrupt handler of interest.

$$\begin{aligned} UltraGreen &\equiv H \\ Green &\equiv AF(UltraGreen) \\ Magenta &\equiv EF(Green) \ \wedge \ EX(handler \notin H) \\ Blue &\equiv AF(Magenta) \\ Red &\equiv \neg EF(UltraGreen) \\ Yellow &\equiv \neg(Red \vee Green \vee Magenta \vee Blue) \end{aligned}$$

A comprehensive explanation of the algorithm and its implementation can be found in [4].

Returning to the control flow graph from Figure 3, we see that 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*.

The entire flow graph of the example program is now *Green*, *Blue*, or *Magenta*. All edges in the graph are annotated with execution cycles; all timing information is taken from Zilog Inc.'s Z86 reference manual, *Z86E30/E31/E40 Preliminary Product Specification*. WCET in the graph can be calculated 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 of 320010 machine cycles between the *Magenta* node at `19` and the interrupt handler at `2C`. The same calculation over the *Blue*

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
Micro00	84	2	Example from ICSE01
ICSE01	55	1	Example from ICSE01
FSE03	35	2	Example for this paper

Figure 4: Benchmark Characteristics

nodes 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, e.g., 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.

5. EXPERIMENTAL RESULTS

The following sections present our experiments applying our tool to a suite of commercially available microcontroller systems. Following these results, we present a narrative of a representative session with our tool, starting from a fresh program, and iterating with our tool until all nodes are either *Green*, *Blue*, or *Magenta*.

Benchmark Characteristics. The benchmarks we have used, see Figure 4, 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, 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 every 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 [5], called "ICSE01," and "Micro00."

Measurements. The results shown in Figure 5 give the final percentages of nodes by color after our tool completed the analysis. For clarity of presentation, we have numbered the interrupt sources in the tables 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 simply 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 our tool rounds percentages down in most cases, or up in the case of percentages less than 1%, so the tables in Figure 5 may not total precisely to 100%.

	Percentage *Green*			Percentage *Blue*		
Prog	IRQ_1	IRQ_2	IRQ_3	IRQ_1	IRQ_2	IRQ_3
CTurk	100%	5%	.	0%	87%	.
GTurk	100%	2%	.	0%	94%	.
ZTurk	100%	2%	.	0%	94%	.
DRop	99%	62%	40%	1%	36%	58%
Rop	99%	66%	37%	1%	32%	60%
Serial	100%	54%	49%	0%	44%	49%
Micro00	56%	45%	.	38%	49%	.
ICSE01	100%	.	.	0%	.	.
FSE03	100%	28%	.	0%	57%	.
	Percentage *Magenta*			Percentage *Yellow*		
Prog	IRQ_1	IRQ_2	IRQ_3	IRQ_1	IRQ_2	IRQ_3
CTurk	0%	7%	.	0%	0%	.
GTurk	0%	3%	.	0%	0%	.
ZTurk	0%	3%	.	0%	0%	.
DRop	1%	1%	1%	0%	0%	0%
Rop	1%	1%	2%	0%	0%	0%
Serial	0%	1%	1%	0%	0%	0%
Micro00	5%	5%	.	0%	0%	.
ICSE01	0%	.	.	0%	.	.
FSE03	0%	14%	.	0%	0%	.

Figure 5: Results With Completed Oracles

		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 6: Adaptive Slicing vs. Fixed k-CFA

Yellow nodes were completely eliminated, and the percentages of *Green* and *Blue* were quite high. The amount of *Magenta* present in the final graphs was uniformly low, less than 2% of the overall graph size on average. Several of the benchmarks had 0% *Magenta* for a given IRQ, which means our tool can safely and completely bound interrupt latency for those particular handlers from anywhere in the program.

Our 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 easily be constructed to run in far less time.

Figure 6 shows the sizes of the graphs generated by the analysis, both with adaptive slicing, and with a fixed k-

Program	Number of Summary Edges			
	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 7: Oracle Information Provided

CFA, where the value for k is fixed to the depth needed by the adaptive slicing.

As mentioned earlier, this technique represents 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 the next section,) anything that causes larger graphs potentially creates more *Yellow* nodes, adding more noise to the output of the tool, and making the entire process increasingly difficult.

Figure 7 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.

Assessment. The complete elimination of *Yellow* nodes from the control flow graphs of the commercial microcontrollers was our primary goal, and this has been accomplished by our tool.

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 brings us much closer to completely bounded latency in 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, gives us high hopes of being able to automatically bound 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 we could expect the total uninterrupted WCET of the *Magenta* cloud to be less than the minimum period of the interfering interrupt handler(s), which would make it relatively easy to reason about these sections with a simple *worst-case response time analysis* [20] or by detailed simulation and testing.

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

6. USER EXPERIENCE

In this section, we detail the complete process of starting with a raw program, and iterating with our tool to add assertions to the time summary oracle until all *Yellow* nodes are eliminated.

We choose one of the moderately sized benchmarks, `Rop`.

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 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 an immediately 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 simple 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 this 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 680 mS.

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 an assertion that we expect later versions of the tool to infer automatically. The letter "[D]" indicates a data-dependent loop which would require a very clever data flow analysis to automatically bound.

```
[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 significantly pared down 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%
```

The user has presented 16 assertions to the oracle, 10 of those based upon manual inspection of the code, rather than external design criteria. The remaining 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 this system's timing behavior has increased as a result of our tool. Testing and simulation can concentrate on the lines of code for which we have provided assertions, and on the *Magenta* nodes, both of which comprise a tiny fraction of the total state space for the code. Our tool produces simple 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.

7. 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, less than 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.

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 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.

8. REFERENCES

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In *Real-Time: Theory in Practice*, Springer *LNCS* 600, pp.74–106, 1992.

[3] G. Bernat, A. Burns, and A. Wellings. Portable WCET analysis using Java byte code. In *Proc. ERTS 2000*, pp.81–88, Jun 2000.

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

[5] D. W. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt driven software. In *Proc. ICSE 2001*, pp.47–56, June 2001.

[6] L. Cardelli. Type systems. In *The Computer Science and Engineering Handbook*, chapter 103, pp.2208–2236. CRC Press, Boca Raton, FL, 1997.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, Jan 2000.

[8] M. Corti, R. Brega, and T. Gross. Approximation of WCET for preemptive multitasking systems. In *Proc. LCTES 2000*, Springer *LNCS* 1985, pp.178–198, 2000.

[9] J. Engblom and A. Ermedahl. Modeling complex flows for WCET analysis. In *Proc. RTSS 2000*, Nov 2000.

[10] J. Engblom and B. Jonsson. Processor pipelines and their properties for static WCET analysis. In *Proc. EMSOFT 02*, Springer *LNCS* 2491, pp.334–348, 2002.

[11] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *Proc. SAS 03*, pp.214–236, 2003.

[12] M. J. Harrold, J. A. Jones, T. Li, D. Liang, and A. Gujarathi. Regression test selection for Java software. In *Proc. OOPSLA 01*, pp.312–326. ACM, 2001.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, April 1999.

[14] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[15] S. Petters and G. Färber. Making WCET analysis for hard real-time tasks on state of the art processors feasible. In *Proc. RTCSA 99*, pp.442–449, 1999.

[16] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proc. OOPSLA 94*, pp.324–340. ACM, 1994.

[17] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proc. ICSE 92*, pp.105–118. ACM, 1992.

[18] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, CMU, May 1991.

[19] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Journal of Real-Time Systems*, 18(2/3):157–179, 2000.

[20] 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, Mar 1994.

[21] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. In *Proc. LCTES 01*, pp.88–93. ACM, 2001.

[22] Z. Xu, B. P. Miller, and T. Reps. Safety checking of machine code. In *Proc. PLDI 2000*, pp.70–82, 2000.