STATIC CHECKING OF INTERRUPT-DRIVEN SOFTWARE

A Thesis

Submitted to the Faculty

of

Purdue University

by

Dennis W. Brylow

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2003

To my parents:
my stepfather, who taught me patience;
my stepmother, who taught me organization;
my father, who taught me humor;
and my mother, who taught me faith –
all qualities I could not have made it
this far without.

ACKNOWLEDGMENTS

This dissertation would not be what it is if it were not for the collected efforts of many friends and colleagues over the years. Several should be thanked here:

- William Rueth of Greenhill Manufacturing who provided me with years of employment before graduate school, a wealth of knowledge and experience, and the use of Greenhill's proprietary software in my experiments.

- Niels Damgaard, my co-author for the ICSE paper, who wrote a great deal of code for the ZARBI Simulator and genetic search algorithm. Also, Wanjun Wang for timely maintenance of JTB and for writing the GUI components for the Simulator.

- Mike Grypp, Phil McGachey, Mayur Naik, Krishna Nandivada, John Regehr, Michael Richmond, and Ben Titzer for proofreading and comments on drafts of this dissertation or its predecessor documents.

- My committee members at Purdue: Doug Comer, Tony Hosking and Jan Vitek, for guidance throughout my graduate career, and especially in the final stages of this work.

- Somesh Jha, my external committee member from University of Wisconsin, for bearing with this process despite the additional overhead that being two states away entails.

- Jens Palsberg: My patient adviser, my Evil Master. A model researcher, a brilliant mentor, and a good friend.

- Petra Eccarius, who provided moral support for uncountably many long nights working in the Lab, and who always knew I could do it.

TABLE OF CONTENTS

Page

LIST OF FIGURES

ABSTRACT

Brylow, Dennis W. Ph.D., Purdue University, August, 2003. Static Checking of Interrupt-Driven Software. Major Professor: Jens Palsberg.

Static checking can provide safe and tight bounds on stack usage and execution times in interrupt-driven systems. This dissertation presents static analysis algorithms and a prototype implementation of those algorithms for statically computing resource bounds in interrupt-driven systems. Advanced knowledge of resource bounds enables real-time system designers to eliminate whole classes of errors from their software before testing begins, thereby reducing the testing effort necessary to achieve confidence in their system.

Despite the ubiquity of hardware interrupts in real-time systems, little prior research has dealt with interrupt-driven software. The benchmark suite of commercially-deployed, interrupt-driven systems examined here includes proprietary Z86-based microcontrollers programmed in assembly language with multiple vectored interrupt sources, a shared system stack, extensive use of unstructured loops, and no formal loop annotations.

The stack analysis bounds the maximum stack size to within one byte of the true maximum in all but one of the programs in the benchmark suite. The deadline analysis found firm worst-case latencies in 30% of the cases; in the remaining 70% of the benchmarks, the prototype reduced the size of the testing problem by an average of 98%. While the testing effort still required for these systems is large, it is several orders of magnitude smaller than the testing problem without deadline analysis.

This dissertation presents novel algorithms for static analysis in the context of interrupt-driven assembly code. The prototype implementation is one of the first tools to incorporate static analysis with testing oracles in an interactive fashion.

1 INTRODUCTION

1.1 Thesis Statement

Static checking can be employed to provide safe and tight bounds on stack usage and execution times in interrupt-driven systems.

1.2 Overview

It was the goal of this research to find a balance between static analyses and design specifications for the purpose of constructing practical development tools in the area of real-time, interrupt-driven software. This effort has been successful; the prototype tool, called "ZARBI" (Zilog Architecture Resource-Bounding Infrastructure), implements novel static analysis algorithms for finding safe and tight bounds on both stack usage and worst-case interrupt latency in the analyzed systems.

The systems analyzed in this dissertation exemplify a class of interrupt-driven software with vectored interrupt handling, unstructured and unbounded loops, limited indirect addressing and limited indirect procedure calls. The benchmark suite includes seven commercial microcontroller systems available to the author, as well as many smaller example programs that demonstrate other interesting real-time programming idioms.

The structure of this dissertation follows the outline below.

- Chapter 1 presents introductory material, the thesis statement, and outlines contributions.
- Chapter 2 describes related work.
- Chapter 3 defines terms and explains concepts that are used throughout later chapters.
- Chapter 4 presents the stack size checking algorithm [14], demonstrating that a control-flow representation containing a program counter, interrupt mask register, and the top stack element at each node is sufficient to bound stack usage in many interrupt-driven systems, as well as check several other safety properties.
- Chapter 5 presents the deadline analysis algorithm used to bound interrupt latency [15]. The ZARBI implementation of the algorithm colors control flow graphs based on interrupt latency, incorporating external timing information into the static analysis in order to bound maximum latencies.

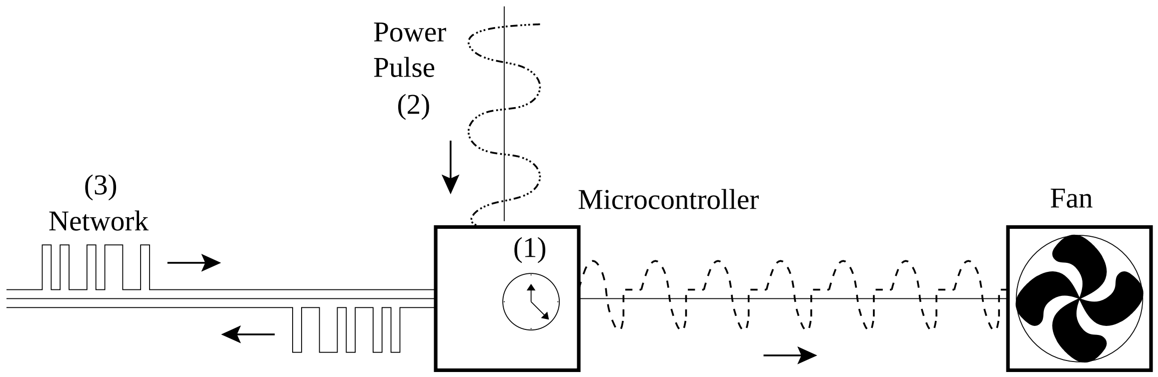


Figure 1.1. Example of an Interrupt-Driven System

- Chapter 6 describes the infrastructure of the prototype system, detailing design choices, algorithmic details, and experiences relevant to building this static checking tool.
- Chapter 7 summarizes and concludes with a discussion of future research directions.

1.3 Embedded Systems

Real-time, reactive and embedded systems are used in applications such as flight control, vehicle management systems, telecommunications, home electronics and medical devices [25,31,40,92]. Many such applications are long lived, interact with their environment continuously, and operate under important real-time constraints. The systems analyzed in this dissertation were designed and marketed with the expectation that they would run for months or years without down time. They are expected to continuously react to input, and failures can potentially cause tangible monetary loss. As the deployment of such embedded systems grows, the need for cost-effective software assurance techniques grows correspondingly.

1.3.1 Interrupt-Driven Software

This dissertation focuses on a common class of real-time systems known as *interrupt-driven* systems. Interrupts and interrupt handlers are used in systems where fast response to an event is essential. Interrupt-driven systems are those in which significant portions of the overall computation rely on interrupts and their handlers.

For example, Figure 1.1, illustrates the operation of one of the microcontroller systems analyzed in later chapters. The example microcontroller has three interrupt sources which interact in a complex fashion. The first interrupt source is an internal timer, used to generate the waveform that controls a bank of variable speed ventilation fans. The interrupt handler regularly recalculates the timer interval to maintain the desired waveform. The second interrupt source is a 60Hz power pulse, which enables the microcontroller to synchronize the waveform output with the fan power source. The first and second interrupt handlers must coordinate with one another in order to correct for phase changes in the fan power, or drift in the internal timer calculations. The third interrupt source is a network communication channel via RS-485 *long-haul* modem, used by a central network controller to poll status, examine sensor readings, and even reprogram the remote microcontrollers. Network communication interrupts can come at virtually any time, and must be given highest priority when they arrive. (The processor manually sequences the RS-485 packets, bit by bit, at the proper baud rate.) If the processing of network traffic takes too long, proper control of the fans cannot be maintained. The full version of the microcontroller system shown here is part of a ventilation system used in an agricultural setting; fan lockup can result in danger to livestock from heatstroke, pneumonia, or deleterious levels of ammonia and methane.

Testing of real-time embedded systems like the example in Figure 1.1 is difficult. While powerful processors can be used for embedded systems, the demand for cost-effective computation results in the use of smaller, resource-constrained devices in far greater numbers [92]. For developers, the reality of resource-constrained devices can mean that the use of convenient, high-level abstractions (e.g., real-time operating systems which provide certain guarantees) is not an option. Software in reactive and real-time embedded systems is often programmed by hand in low-level languages like C and assembly [25]. Real-time software can rely heavily on hardware interrupt handling, have no high-level process model, and leverage little or no compiler assistance – all factors which can make analysis of the software more difficult. Without the high-level abstractions most software analysis techniques depend upon, such systems are often evaluated for safety and correctness only through extensive testing or simulation.

1.3.2 Testing

Component and integration testing of embedded systems can be intensely time-consuming, prohibitively expensive, and is often less than comprehensive. Unlike software on general purpose computing platforms, embedded systems are hard to instrument. Embedded systems have narrow information channels: internal register states are difficult to access externally without altering the system; hardware interactions are difficult to manipulate without distorting key timing properties of the system; and finally, resource constraints usually render on-chip monitoring infeasible [42].

Testing of *real-time* embedded systems is even more difficult than embedded systems, because the real-time components of the software add nondeterminism to the system. Small variations in the interrupt requests caused by external triggers and internal timers can result in different behavior between runs even if the controller is executing the same computation on the same data.

Furthermore, even if it were practical to ascertain precise machine state from embedded systems, the number of possible execution paths increases combinatorially in a interrupt-driven system. For any given machine instruction in a segment of code where interrupts are enabled, control could potentially pass either to the next instruction, or to any of the enabled interrupt handlers. In this way, the number of transitions in an equivalent state machine for a interrupt-driven system increases exponentially in the number of available interrupts. Traditional coverage testing quickly becomes intractable in this setting.

Consider the example system from Figure 1.1: assume that the first interrupt source (internal timer) is triggered 180 times per second, with the handler executing for 100 microseconds; the second interrupt source (power pulse) occurs 60 times per second, with the handler executing for 10 microseconds; and the third interrupt source (network traffic) occurs once per second, and takes 100 microseconds to handle. If these events take place completely independently, then the odds of observing all three handlers conjunctively contributing to the maximum stack height in any given microsecond time-slice are roughly 1×10^{-9}. This worst case behavior could be expected once in a billion observations, assuming it was even possible to gather stack height data from the embedded system and that normal test inputs would even explore that corner of the problem space. Nevertheless, if 1000 such systems are deployed and operated for years, it is a near-certainty that this unusual worst-case behavior will occur in the field. Further note that these probabilities are for a greatly simplified example system.

While it is unlikely that the need for full-scale testing will ever be completely supplanted by any other methodology, there is great potential for software verification tools to substantially decrease both the time and effort for testing real-time systems. For example, static analysis of timing properties in real-time systems could eliminate whole classes of errors prior to testing.

1.3.3 Practical Challenges

The example in Figure 1.1 is a simplification of a real-time system actually in production; the processor is an 8-bit Z86 microcontroller, with 256 bytes of RAM, 4K of program ROM, and a 12MHz clock [100]. The software for the example system was written by hand, in Z86 assembly language, and is about 2500 lines of code, with comments. The prototypes of this particular system underwent months of testing prior to actual production. The final production model did not include the RS-485 network hardware; even though the software was written to handle the network connection, production deadlines did not allow sufficient testing to determine what adverse impacts, if any, could be expected when the interrupts interacted.

Several pressing questions prevented the deployment of the network functionality for the example controller.

- How high would the stack grow if the controller's network communication handlers were triggered during normal operating modes? The memory layout of the Z86 does not offer hardware protection for global data registers from the system stack; if the stack grew larger than the designers had anticipated, it would overwrite other data registers.

- How would the network communication handlers interfere with the other interrupts in the system? Could a network packet cause the controller to miss one of its deadlines for generating the proper control signals? Could the other interrupts cause the network communication interrupt to miss its deadline for properly interpreting a network packet?

Worse yet, even *without* the network interrupts enabled, it was not clear that the controller would necessarily meet all of its deadlines.

The ZARBI tool was designed to help address questions like the ones above when analyzing interrupt-driven systems. The static analyses presented later in this dissertation produce safe, tight bounds on stack usage and interrupt latency. With these bounds in hand, system designers can avoid much of the costly testing effort that would otherwise be required to determine whether or not the system has sufficient resources.

1.4 Contributions

The production microcontrollers studied in this dissertation rely on vectored, asynchronous interrupt handling to accomplish their work. They use a limited form of indirect procedure call, extensive `goto`-like "`JMP`" instructions, and have no formally annotated loop bounds. The goal of this research project was to devise automated techniques for producing accurate bounds on resource consumption in interrupt-driven systems. A side benefit of the research was the construction of a prototype tool capable of providing resource bounds for the kinds of systems exemplified by the characteristics above.

The primary contribution of this work is ZARBI, the Zilog Architecture Resource-Bounding Infrastructure. The prototype tool computes conservative, tight bounds on stack usage and worst-case interrupt latency for interrupt-driven systems written in Z86 assembly language. These bounds allow the system designer to eliminate whole classes of errors from the software before testing even begins, thereby reducing the testing effort necessary to achieve confidence in the system.

Secondary contributions of this work include novel algorithms used in the core of ZARBI to bound stack height and maximum interrupt latency, respectively. This is the first such work on tractable control-flow analysis in the presence of vectored interrupt handling.

Additional analyses also check for several classes of semantic errors in the Z86 program, including using simple types to detect stack manipulation errors. In addition, ZARBI contains components for enhanced visualization and debugging of control-flow graph cycles during the interactive process of interrupt latency analysis.

2 RELATED WORK

In the general case, the problem of bounding stack sizes and maximum execution times is equivalent to the halting problem [84]; 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 [56], 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 [80], which uses the following conditions to guarantee decidability:

- No asynchronous interrupts
- No recursion
- No indirect calls
- No `goto` instructions
- Strictly bounded loops

In the 1990's, researchers have 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 Source-Level Timing Schemas

In 1989, Alan Shaw wrote, "When interrupts are permitted and both interrupt handling times and frequencies are bounded, the effects of processor sharing between a user process and one or more interrupt handlers can be included in a timing analysis," [89]. While this is certainly true, it remains very difficult in practice to automatically ascertain interrupt handling times. Interrupt frequencies are entirely beyond the scope of automated program analysis, and generally fall under the category of design criteria for a given system. Shaw's *timing schema* for high-level languages, (by which he meant Algol,) has served as the basis for over a decade of subsequent research on analyzing maximum execution time for software. On the topic of interrupts, Shaw indicated that the system could be extended to account for interrupts using the equation,

$$t'_{max}(S) = t_{max}(S)/(1 - f_{max} \times t_{max}(IH))$$

where $t_{max}(S)$ if the uninterrupted, *straightline* maximum execution time for statement S, $t'_{max}(S)$ will be the maximum execution time of statement S taking interrupts into account, and f_{max} and $t_{max}(IH)$ are the known interrupt frequency and interrupt handler execution time. Shaw concluded that, "timing predictability seems impractical when a process can be preempted at arbitrary points in its code," and left the matter at that. A large body of work has stemmed from this original premise, as exemplified by papers like Lim et al. [48], which extend Shaw's basic timing schema to account for features of modern processors such as pipeline, data cache, and instruction cache effects. Engblom et al. [28] concentrate on co-transformation of source-level schema in order to inform analysis of compiler-optimized object code. All of the work listed above assumes an absence of interrupts, or trivially isolatable interrupt behavior, in spite of the fact that virtually all modern processors used in real-time systems have vectored interrupt handling facilities, and all real-time systems known to the author have made use of those facilities.

2.2 The False Path Problem

In 1996, 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 – branches that aren't taken, interrupt handlers that aren't yet enabled, etc. 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 [54]. Altenbernd used symbolic execution to track possible values of key conditional variables, and thereby pruned infeasible paths out of the control-flow graph. This is essentially the same technique used by ZARBI to prune away a substantial number of infeasible interrupt handler paths from the control-flow graphs.

2.3 Higher-Level Languages

Liu and Gómez [50] automatically transformed Scheme code directly into time-bound functions, based upon partially-known input structures. They then plugged in numbers gleaned from intensive profiling to approximate the actual execution time of the compiled code. Their method has fared well initially, yielding execution time estimates very close to measured execution times. However, the source language and its accompanying transformations have no provisions for vectored interrupts, and the technique glosses over issues concerning accurate low-level timing of primitives by averaging together tens of millions of runs of representative code. It remains to be seen whether the need for first-class lambda expressions will outweigh the need for accurate low-level timing in the community of real-time system designers.

Applying essentially the same concept as [50], but at a lower level, Lundqvist and Stenstrom [51] augmented a PowerPC simulator to use an "unknown" value. The unknown value allows a variant of instruction-level simulation, without having to know precise input. In addition, they used path-merging heuristics to maintain a tractable number of paths. Their work does not consider vectored interrupt handling, although their path-merging technique may be generalizable to assist in keeping paths caused by interrupt handlers to a manageable number.

Research on Real-Time Java [12] aims to make Java a legitimate language choice for real-time programmers. While the object-oriented programming model has little in common with Z86 assembly language, work on RTJ addresses many of the same problems as this dissertation. Implementation of *scoped* memory for RTJ [9] addresses issues of bounding memory allocation, and more importantly, bounding execution time impacts of memory management. Others have worked on WCET analysis for Java Byte Code [8] and portable WCET annotations for Java Byte Code [10]. However, the very abstractions that make Java an attractive development environment hamper accurate analyses; just getting back the *gain time* lost to overestimation of WCET due to dynamic dispatch is a difficult problem [43].

2.4 Special-Purpose Languages

Many special-purpose languages have been created for use in real-time systems. Real-Time Euclid [46] has provisions for schedulability analysis built in – all loops have a bounded number of iterations or execution time.

ESTEREL [11] is a prime example of a synchronous language that can be used for programming reactive systems. Synchronous languages use *instant broadcast* between processes, which means that interprocess communication and other data handling take an irrelevantly small amount of execution time. While synchronous languages are well-suited to purely reactive systems, they are not as well-suited to interactive or transformational systems. The embedded systems examined in this dissertation exhibit characteristics of all three kinds of systems: reactive, interactive, and transformational.

Like synchronous programming languages, the Giotto project [40] seeks to provide a platform-independent abstraction for programming real-time systems. Giotto programs are concerned with functionality and timing properties of the system. Tasks are organized into modes, and communicate with one another through drivers – underlying code for transporting data between processes, sensors, and actuators. The actual tasks and drivers are not implemented in Giotto; they are executed in a platform-dependent fashion using compilers that must conform to Giotto's constraints in order to guarantee that the final system meets the properties promised by the Giotto model.

Extending the Giotto project, the E-Machine [41] is a platform-independent virtual machine that supervises the timing of a real-time system with respect to the external environment.

2.5 Preconditions for Success

As mentioned in the previous section, Puschner and Koza codified the standard conditions for making WCET analysis tractable in 1989 [80]. These limitations were no interrupts, recursion, indirect calls, or `goto`'s, with a-priori bounds on all loops.

Nine years later, a survey paper on techniques for static analysis of embedded software [52] assumes all of these preconditions except for the `goto` rule. The brunt of work in the WCET area continues to revolve around timing effects caused by cache misses. Cache effect analysis is not applicable to many real-time systems, like the Z86 family of processors, which do not even have cache memory.

Li and Malik's Cinderella project [47], so named for the fictitious girl's hard real-time constraint with respect to midnight and pumpkins, automatically formulates WCET analysis as an integer linear programming problem. Their tool analyzes source code for the Intel i960KB processor, and locates critical variables with respect to the timing analysis. The user then manually assigns bounds to the critical variables, and the analysis calculates final execution times. Cinderella operates under the standard Puschner and Koza assumptions, and does not allow interrupts.

Work in automatic detection of induction variables [62], and bounding of *unnatural* loops in low-level languages [38] is applicable to loops present in the commercial microcontroller systems examined later in this dissertation. Healy and Whalley's approach [39] 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.

2.6 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 [75, 95].

The FLAVERS system at University of Massachusetts, (FLow Analysis for VERifying Specifications), is a flexible framework for flow analysis of concurrent programs [23, 65]. FLAVERS has even been extended to analyze infinite executions [66], 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 [79]. The algorithm presented later in this dissertation differs from Podelski [79] 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.

Analysis of partially-implemented real-time systems [7] is tangentially related to this dissertation, in that the Z86 simulator in ZARBI models the unimplemented portions of systems for test purposes, and static analysis of timing bounds may involve modeling external inputs in a similar fashion.

Research at University of Wisconsin has used graph reachability [83] as a mechanism for program analysis. Context-sensitive analysis of the sort employed by ZARBI has been shown to be undecidable in the general case [84], as it is equivalent to Post's Correspondence Problem. Fortunately, the straightforward heuristic that stack sizes in Z86E30 software can be no larger than the meager 256 bytes of total RAM gives the ZARBI algorithm decidability.

Maximum execution time is formulated as a graph theoretic problem in Puschner and Schedl [81], using T-graphs. T-graphs are substantially similar to the control flow graphs used in ZARBI with edges weighted by execution times. Relative capacity constraints provide information about infeasible paths in the T-graphs using information provided by the user. When the T-graph construction is complete, the search problem is passed on to an integer linear programming (ILP) solver. The T-graph approach allows goto statements and can provide precise maximum execution time – rather than execution time bounds – in cases where every instruction takes an invariable amount of time to execute under all circumstances.

Like Brylow et al. [14], Wegener and Mueller [98] shows that static analysis and evolutionary testing can be used successfully in concert to seek both upper and lower bounds on worst-case execution time.

2.7 Type Theory

Advances in the static analysis of programs have addressed a plethora of safety issues, including bounding resources like stack size.

Palsberg and O'Keefe [74], and Palsberg and Schwartzbach [76] present and prove soundness for a type system that checks the safety of a calculus with untyped lambda terms. This is essentially the same kind of safety problem as type checking the basic stack operations in Z86 programs, and a similar type system is used by the ZARBI stack-bounding analysis to catch several classes of potential errors.

In 1996, Necula and Lee proposed a technique for embedding a formal proof of correctness in code [69]. In 1997, Necula refined his *Proof-Carrying Code* mechanism [67], and showed what such a framework might look like. In 1998, Necula and Lee revealed a working, non-trivial implementation of the PCC concept [68]. The proof-carrying code concept includes annotations for loop invariants, which could ultimately be helpful in WCET analysis of loops.

Morrisett's TAL [59] is a RISC-like assembly language, with annotations at basic block and allocation points that allow the code to be proven type-safe. In this way, typed assembly language is a particular kind of proof-carrying code, with the overhead of the proof being dramatically reduced. Extensions to TAL include type-safe stack management [58] for a substantial subset of the Intel x86 instruction set [57]. Another extension to the TAL system is Crary and Weirich's type system for bounding resource consumption, particularly time bounds [22].

The tool presented in [99] checks SPARC machine code for memory safety using type state checking and input annotations. This approach has benefits similar to [68] and [59], in that safety checking is done at the lowest level, and does not entail trusting an optimizing compiler. Also like [68] and [59], the systems presented in [99] was not designed with analysis of timing properties in mind.

While all of the papers above present valuable techniques for static analysis of low-level programs, none allow for preemptive interrupts of any kind.

2.8 Tools

The Advanced Software Technology (ASTEC) group centered at Uppsala University has built a substantial infrastructure for analysis of WCET in real-time systems [29]. The ASTEC group 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 [27] to describe feasible execution paths, or facts about low-level factors like pipeline effects on the execution time [30]. 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. WCET can then be estimated using the function

$$WCET = maximize(\sum_{\forall entity} x_{entity} \times t_{entity})$$

where x_{entity} is the execution count for each entity, t_{entity} is the known execution time of each entity, and flow constraints ensure that the system examines only feasible paths [27]. The research at ASTEC is in concert with IAR Systems, and therefore has been tested at several points against realistic real-time systems. Work on the

ASTEC infrastructure continues, with support now included for flow analysis of C programs [35].

The University of Saarland Embedded Systems (USES) group has used abstract interpretation [21, 70] and ILP solvers to extensively model the Motorola "ColdFire" MCF 5307 processor [31]. 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 [31].

Commercial ILP solvers like CPLEX [44] and `lp_solve` [72] have been employed to analyze advanced processor features like cache and pipeline analysis [3, 32], and branch prediction [55].

2.9 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 this dissertation have interrupt handlers heavily integrated with the main program; they share the same system stack, operate on the same relatively small set of registers, 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.

Chapters 4 and 5 present techniques for analysis of interrupt-driven programs that mitigate much of the exponential increase in state-space during analysis.

3 FRAMEWORK

The next several chapters present the static analysis techniques used to bound stack size (Chapter 4) and execution time (Chapter 5) in interrupt-driven software. This chapter defines common concepts and abstractions used throughout chapters 4 and 5, as well as in the chapter on implementation details (Chapter 6).

3.1 Control Flow Graphs

The algorithms presented in this dissertation operate on an abstraction of program states known as a *control flow graph* [2]. This section defines flow graphs and terminology that will be used in subsequent discussions of the algorithms.

A *control flow graph* is an abstraction of program states and the transitions between them. Details and examples of control flow graph construction are given in sections 4.2, 5.2, and 6.1.

A control flow 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*. For the analysis algorithms presented later in this chapter, a control flow graph (abbreviated hereafter as *CFG*) is the first component of 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.

A control flow graph, (abbreviated hereafter as *CFG*,) is a *digraph* [87], 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. A vertex v_0 is *upstream* of v_k if there exists a path from v_0 to v_k, but not vice-versa.

Resource-bounding algorithms deal extensively with paths in the CFG. A *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* [87] 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.

The resources to be analyzed in a CFG are represented as edge weights. The weight function w maps each edge to an integer cost. G is therefore a *weighted digraph*, or *network* [87]. 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 $\forall i \in \{0, ..., k-1\} : w(v_i, v_{i+1}) = 0$.

Many of the algorithmic details of resource bound analysis in this dissertation 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 problem is a classical graph problem [87] equivalent to many problems in static analysis. The *longest path* in the graph is defined as the path with the largest cost, which is not necessarily the path with the largest number of edges.

3.2 Stack Size Analysis

This section presents properties that a CFG may possess. Later chapters will show how a stack size analysis algorithm can take advantage of these properties. For stack analysis, the weight function w is defined to associate each edge in the graph with an integer change in stack height. In the resulting weighted digraph, stack size analysis is equivalent to the search for a longest path rooted at vertex *terminus*.

In the general case, the longest path problem is known to be NP-hard and thus is considered intractable [87]. However, the control flow graphs examined here have additional structure that can be exploited to provide a more efficient analysis. The next several subsections outline properties that make a CFG more amenable to stack size analysis.

3.2.1 Negative Cycles

In the algorithms presented later, 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, such programs are not dealt with in this dissertation. Negative cycles can be detected in $O(V^3)$ using Floyd's Algorithm [87].

3.2.2 Summary Edge Closure

Later algorithms will use the concept of summary edges, as defined below. A *summary edge* e_Σ 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 3.1. 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

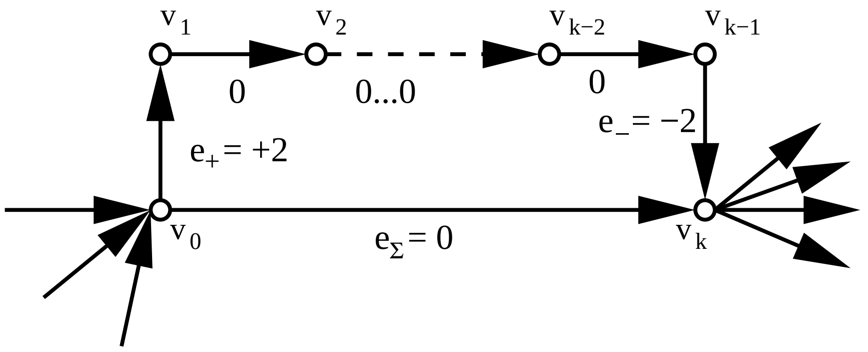


Figure 3.1. Summary Edge Closure

correspond to the type-checking of stack elements specified in Section 4.3.4, which 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. Construction of summary edges is explored in greater depth in Section 4.3.

A graph with no negative cycles can be closed with respect to summary edges in time polynomial in V [53].

3.2.3 Positive Cycles

In this dissertation, the longest path in a CFG is not defined for graphs with positive cycles. 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.

3.2.4 Null Cycles

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 [87].

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, $O(V)$ [87].

3.3 Stacks and Contexts

Some of the algorithms and techniques presented later in this dissertation cannot be understood solely in the context of control flow graphs without additional program analysis concepts. This section defines terminology that will be used in later discussions.

A *stack* is a last-in, first-out data structure [71]. 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 "$\{\}$".

An abstraction used in many programs is the *procedure call*, in which a common segment of code is factored out into a *procedure* or *subroutine*, which can then be *called* from multiple program locations [2]. The program points from which procedures are called are termed *call sites*.

A CFG that differentiates the vertices for the same procedure when called from different call sites is *context sensitive* [70]. Context sensitivity necessitates representing additional state information at vertices in the graph. Because a procedure A can call another procedure B before completing, the context required to distinguish two states in the program may require more than one call site. Context represented as a stack of call sites is a *call string* [70, 88].

With call strings comes 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.

3.4 Refinements

Conceptually, there are two mappings required to get from a raw program to resource bounds. The first mapping is from the program to the CFG. The second mapping is from the CFG to the resource bounds. The previous sections in this chapter have concentrated on the second mapping. This section concentrates on the first mapping – translating a raw program into a precise and compact CFG.

The next several subsections present concepts underlying the construction of compact and precise graphs for resource bounding analysis. Further details can be found in [2, 70].

3.4.1 Graph Building

Naïve CFG construction algorithms can result in a combinatorial explosion of the vertex state space. It will be important later to optimize the size and complexity of the graphs.

At one end of the spectrum, consider a CFG representation where every vertex in the graph contains the values of every binary digit of state stored in any variable used in the program. The *precision* of this representation is very good, because every possible state of the program can be unambiguously differentiated from every other. However, the size of the state space for vertices in the CFG is exponential in the number of bits of storage, resulting in very large graphs even with small programs.

At the other end of the spectrum, consider a CFG representation where each vertex of the graph represents a particular executable instruction in the program. Such a representation is compact, being linear in the size of the program. However, because such a CFG lacks context sensitivity, it may contain many unrealizable paths, and thus lacks the precision required to give useful resource bounds for any of the programs examined in this dissertation. Section 4.1.1 revisits this discussion in the context of a specific hardware architecture.

Later chapters will show that for practical reasons, an implementation must find a middle ground where the CFG has enough precision to accurately model the resources that must be bounded, without the size of the graph becoming unmanageable.

3.4.2 Demand-Driven Construction

For the algorithms presented in this dissertation, there is no need to represent program points that cannot be reached by any execution path. Unnecessary expansion of the CFG can be avoided by constructing the graph in a *demand-driven* fashion, where portions of the graph will only be constructed when they are known to be needed according to a given criteria. An example of this is to build only the CFG containing program states that are reachable from the *terminus* vertex.

The algorithms presented in later chapters also do not need CFG's to represent unrealizable interrupt paths. *Abstract interpretation* can be used to approximate values without completely simulating a program [21]. Later chapters show that by approximating the contents of certain control values in the hardware, many unrealizable interrupt paths can be omitted from the CFG's.

3.4.3 Avoiding False Paths

As alluded to in Section 2.2, model precision can be increased by avoiding *false paths* in the CFG. A false path π_{false} is defined as a path for which the sequence of vertices corresponds to a sequence of states that cannot occur, either because the sequence would violate the semantics of the program, or does not correspond to what the hardware does.

One of the techniques available for curtailing false paths is to model only realizable paths using call strings [88]. Call strings introduce context-sensitivity to CFG construction, which is both more precise and more expensive to calculate [70]. The disadvantage of this technique is that the allowable state space of vertices in the graph increases exponentially in the number of bits required for the call strings.

3.4.4 Adaptive slicing

While arbitrary length call strings add precision to CFG's, the size penalty can greatly increase the complexity of building the graph. A trade-off can be made between precision and size by using *call string suffixes* [88], with which only the topmost n elements of the call string are stored, for some limiting value of n.

Varying the value of n in the CFG allows the degree of stack context to be adjusted for the precision required for analysis. In this way, additional context can be stored in vertices that are otherwise difficult to analyze, while more compact call string suffixes can be used in graph segments requiring less precision.

A graph with variable length call string suffixes is *multi-resolution*, indicating that the amount of context at vertices can be varied according to space and precision concerns. The technique of adding more detail to a static analysis only where it is required to reach desired precision is described in [78].

CFG cycles caused by insufficiently long call string suffixes can be detected in time polynomial in V, as described in Section 6.3.2. Sections 5.2.4 and 6.3.2 present the adaptive slicing technique used for constructing multi-resolution CFG's, and Section 5.3.2 discusses the precision/space trade-off of multi-resolution analysis.

3.5 Deadline Analysis

Deadline analysis of CFG's is similar to stack size analysis, but the CFG's have different properties. The weight function w is defined to associate each edge in the graph with a positive integer execution time count. Like stack analysis, the final deadline analysis graphs do not contain cycles. Unlike stack size analysis, deadline analysis CFG's are searched backward for longest paths ending at vertex *terminus*, rather than starting at *terminus*. Deadline analysis graphs do not need to be closed with respect to summary edges because w is defined to provide only positive, non-zero edge weights.

The problem of searching for longest paths ending at a given destination vertex in a digraph is the *multi-source* longest path problem and can be solved for acyclic digraphs in linear time [87].

Chapter 5 presents methods for identifying, bounding, and eliminating positive cycles in the initial deadline analysis control flow graphs.

3.5.1 Time Summary Edges

A key problem in deadline analysis is that many programs do not naturally correspond to an acyclic CFG. In the experiments presented later in this dissertation, none of the benchmark suite of test programs corresponded to an acyclic initial CFG.

Cycles are common in deadline analysis CFG's because positive cycles 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 algorithms shown later do not define the longest path in CFG's with positive cycles.

Loops that produce positive cycles in CFG's may have bounds that can be determined by other types of analysis. Section 5.4 gives examples of loop constructs in real programs that can be bounded through methods other than static analysis.

Given a positive cycle π_{cycle} and a maximum cost bound C_{max} that has been determined by other methods 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 3.2.

In order for the deadline analysis algorithm to remain conservative, time summary edges must be *admissible* [86]. 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 arrive at correct bounds.

Time summary edges cannot be used to summarize cycles in all cases; later chapters will discuss the types of program loops that can be eliminated with time summary edges. Section 5.2.3 describes the use of time summary edges in CFG construction, and Section 5.3.2 presents results of an empirical study of time summary edges required for real programs.

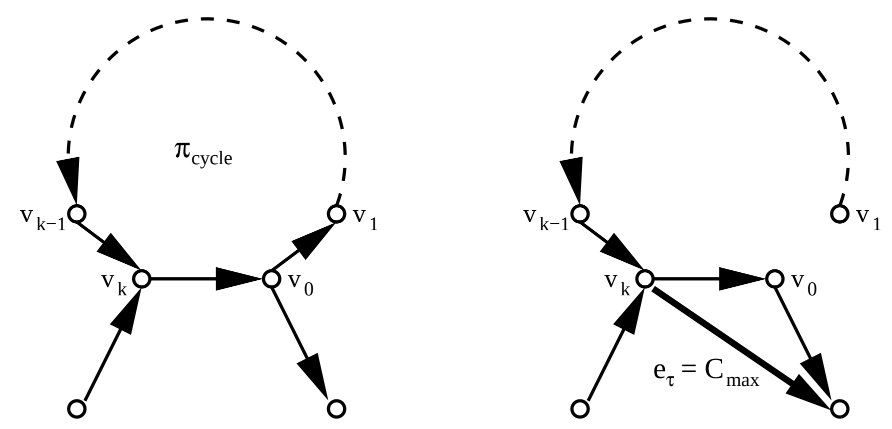


Figure 3.2. Time Summary Edge

4 STACK SIZE ANALYSIS

Static analysis can provide safe and tight bounds on stack usage for interrupt-driven systems implemented on the Zilog Z86 platform.

This chapter presents in detail the overall problem of stack-size analysis in such systems, the algorithm used in ZARBI's analysis, and the results of applying this tool to a suite of commercial embedded systems.

After a brief overview in section 4.1, section 4.2 presents a small example of an interrupt-driven program and its flow graph. Section 4.3 describes the algorithms used in this dissertation to find bounds on stack sizes, and section 4.4 shows experimental results produced with ZARBI. Section 4.5 summarizes the chapter and evaluates the prospects for scaling up these techniques to other processors, such as the Motorola 68000 family.

4.1 Overview

As mentioned earlier, resource-constrained devices are used in many applications. Examples include cell phones, personal digital assistants, digital thermostats, and many others. While larger processors can be employed to comfortably implement embedded systems, economic realities result in the deployment of cheaper processors with tighter resource constraints. It can be difficult to fit required functionality into such a device without sacrificing the simplicity and clarity of the software.

The focus of this dissertation is on small, interrupt-driven devices based on the Z86E30 processor [100], a descendant of Zilog's Z8 processor. The Z86 features 256 8-bit registers, 4K of instruction ROM, and 24 I/O lines organized into three 8-bit ports. In addition, the Z86 has six levels of vectored interrupt processing, and two internal timers. Despite the Z86's limited resources, it is deployed in many elaborate systems where larger, more powerful processors are not cost effective. In many such systems, the Z86's RAM space, ROM space, and I/O lines are pushed to the limit. One of the proprietary embedded systems we have examined has a single Z86 phase-controlling three variable speed fans, operating five heating/cooling units, watching four temperature sensors, monitoring 60–cycle power for brown-outs, networking with a system overseer via RS–485 serial port, and displaying all of its readings on an intelligent LCD unit, all in real time. In such applications, the software is often manually optimized in assembly language, to squeeze every byte out of the ROM, and to use every available register of RAM.

Other processors used for embedded applications comparable to the Z86 include derivatives of the Motorola 68000 series [60]. For example, Palm Pilots and their clones are based on the Motorola DragonBall CPUs (MC68328 [61]), and some cell phones are based on the same architecture family. These processors have maskable, vectored interrupt handling much like the Z86. Devices such as Palm Pilots and cell phones, which function primarily by processing external inputs, can use vectored interrupt handling to provide prompt responses.

Compared with the 68000's, the Z86 has a much smaller instruction set and fewer interrupts (6 interrupts versus 18 in the case of DragonBall MC68EZ328). Yet the Z86 is capable of vectored interrupt handling, making it attractive for rapid prototyping of programming tools.

The dissertation presents algorithms that have been designed and implemented to assist developers with three tasks that can consume a significant part of a real-time system programmer's time:

- Stack-Size Analysis: On the Z86, the stack exists in the 256 bytes of register space, and it is critical that the stack does not overflow into other reserved registers, corrupting data used elsewhere in the program. At the same time, overestimating the stack requirements takes away badly needed registers. The algorithm given later in this chapter finds safe and tight upper and lower bounds on the maximum stack size for all but one of the test programs examined.

- Type Checking of Stack Elements: Items are taken off the stack either with a POP instruction, or when returning from a procedure or an interrupt handler. The analysis presented in this dissertation uses an implicit type system with just four types – interrupt information, code address, interrupt mask, unknown – and checks that the data on top of the stack has the right type at the appropriate time.

- Interrupt-Latency Analysis: The microcontroller systems examined need to handle interrupts within hard real-time bounds. Chapter 5 presents techniques for finding upper bounds on interrupt latencies.

While the overall analysis of these embedded systems requires domain-specific knowledge about the applications, the tools presented in this dissertation accept as input the bare, unannotated Z86 assembly code.

ZARBI's stack size bounding functionality is based on a known algorithm for model checking of pushdown systems [79]. That algorithm is closely related to the style of interprocedural analysis for C that has been studied by Reps [83]. However, the presence of vectored interrupt handling creates additional challenges, as explained next.

4.1.1 The Stack Size Problem

Given a program in Z86 assembly language, the stack size checking algorithm first builds a control flow graph (as previously defined in Section 3.1), and then runs the desired analyses on the CFG. The key question in this concerns the way to abstract a Z86-machine state into a node in the CFG:

How much of a Z86-machine state should be represented in a CFG node?

In one extreme, a node contains the whole Z86-machine state. Such a flow graph would be huge, that is, in the worst case, about $2^{256*8} = 2^{2048}$ nodes. It is beyond current means to represent that many nodes.

In the other extreme, a node represents just the program counter (PC). Such flow graphs are useful for interprocedural analysis of C programs [83], yet they are of little value in the presence of vectored interrupts. When control transfers to an interrupt handler, the current address is placed on the stack, and all interrupts are disabled. If one does not model the interrupt mask register (IMR) in which it is recorded whether interrupts are enabled or disabled, then the analysis is led to believe that a new interrupt can occur as soon as control has arrived at the handler. This process can be repeated, with the result that the stack, seen from the analysis's point of view, can grow without bounds.

There is another consequence of not modeling the IMR; if an interrupt request arrives at a given execution point it cannot be guaranteed that the request will be handled within a finite amount of time. The core of the problem is that false interrupt handler paths may appear in the graph if the IMR value is not approximated.

The above observation makes it clear that the stack size checking algorithm needs to model at least some of the IMR. On the Z86, the IMR consists of seven bits, of which one is the *master bit* which enables or disables all interrupt processing, and six others enable or disable individual interrupts [100]. An interrupt will only be handled if both the master bit and its own bit are set. When an interrupt handler is called, the master bit is automatically turned off. If an interrupt is not handled as soon as it arrives, it will wait (in the IRQ register) until the IMR changes to a value that entails that the interrupt can be handled.

One could consider modeling the PC and the master bit of the IMR. However, this is just as troublesome as modeling only the PC, as one of the tasks of an interrupt handler often is to re-enable interrupts by turning on the master bit. When this happens in the interrupt handler itself, the analysis is led to believe that an interrupt for that same handler can now occur exactly at the point of setting the master bit, leading to a stack growing without bounds, as above.

The CFG used for stack size analysis therefore models the PC and the IMR in their entirety. A Z86-assembly program is typically on the order of 2^{10} lines of code (because there is 4K of instruction ROM), and the IMR is seven bits, so an upper bound on the number of nodes is $2^{10+7} = 2^{17}$. Because of the six interrupt sources, each node in the flow graph can have up to six edges going to interrupt handlers, and one or more edges corresponding to non-interrupt operation. This means that the graph is likely to be less sparse than often seen in program analysis of C programs. It may be possible to model some abstraction of the PC and the IMR, thereby reducing the overall size of the state space, but this idea is not explored by this dissertation.

The CFG can model more than the PC and the IMR, but it is not clear in general which other registers it would be beneficial to model. Chapter 5 describes the addition of stack information to each node in order to refine the model. The next key question is:

> *Can modeling just the PC and the IMR be sufficient for a useful programming tool?*

In other words, can the modeling of the PC and the IMR be a good middle ground between modeling the whole machine and modeling the PC? The criteria for usefulness in this context are given by

- the degree to which the resultant CFG is a good basis for the three kinds of checks that the tool should support: stack-size analysis, type-checking of stack elements, and interrupt-latency analysis; and
- the amount of time and space it takes to build the CFG and perform the checks.

The remainder of this chapter presents an experimental evaluation of the above question.

4.1.2 Results

The stack size checking algorithm presented here is able to produce tight, safe bounds on maximum stack usage for six of the seven proprietary embedded systems, as well as a number of other interesting test inputs. In addition, the CFG's constructed are annotated with information about time, space, safety, and liveness, which allows verification of several code safety properties. The stack size estimation technique presented in this chapter is one of the first to give an efficient and useful static analysis of assembly code, and appears to be the first to use symbolic execution over an interrupt mask register to produce a tractable flow graph in the presence of vectored interrupts.

The prototype implementation includes a Z86 simulator, which has provided lower bounds on the maximum stack sizes, against which the upper bounds can be compared.

In six of the seven commercial cases, and for all of the additional test input cases, the algorithm gives an excellent estimate of the maximum stack size. In all cases, this estimate was either exact (that is, equal to the lower bound that we found via simulation), or at most two bytes more than the lower bound.

For the seventh commercial case, the stack size cannot be bounded without a more detailed analysis including either explicit loop bounds, or enough data flow information to infer loop bounds.

Also in six of the seven commercial cases, the type-checking algorithm was able to check the types used in all stack manipulations, and found no errors. The seventh case could not be checked, because the stack bound must be known for the type-checking algorithm to succeed. Several additional test inputs were created with deliberate

```
; Constant Pool (Symbol Table).
; Bit Flags for IMR and IRQ.
IRQ0 .EQU  #00000001b
; Bit Flags for external devices
; on Port 0 and Port 3.
DEV2 .EQU  #00010000b

; Interrupt Vectors.
     .ORG  %00h
     .WORD #HANDLER  ; Device 0

; Main Program Code.
     .ORG  %0Ch
  INIT:              ; Initialization section.
0C   LD   SPL, #0F0h ; Initialize Stack Pointer.
0F   LD   RP,  #10h  ; Work in register bank 1.
12   LD   P2M, #00h  ; Set Port 2 lines to
                     ; all outputs.
15   LD   IRQ, #00h  ; Clear IRQ.
18   LD   IMR, #IRQ0
1B   EI              ; Enable Interrupt 0.
```

Figure 4.1. Example Program (part 1)

stack manipulation errors; all errors were caught by the prototype implementation of the algorithm.

In summary, by modeling only the PC and IMR registers, the stack size checking algorithm is able to provide solid stack-usage bounds for six out of the seven real-time systems. The analysis is sufficiently fast and precise to be useful in practice. However, providing stack-usage bounds for the seventh system, and execution time bounds in general, requires modeling of additional information, as discussed in chapter 5.

4.2 Model Building

This section gives an informal presentation of concepts that will be rigorously defined in section 4.3. Figures 4.1 and 4.2 show a small Z86 program featuring a main program loop, and a single interrupt handler, both of which can call a shared procedure. Figure 4.3 shows the corresponding flow graph.

```
  START:              ; Start of main program loop.
1C   DJNZ r2,  START ; If our counter expires,
1E   LD   r1,  P3    ; send this sensor's reading
20   CALL SEND       ; to the output device.
23   JP   START

  SEND:               ; Send Data to Device 2.
26   PUSH IMR        ; Remember what IMR was.
  DELAY:
28   DI              ; Mustn't be interrupted
                     ; during pulse.
29   LD   P0,  #DEV2 ; Select control line
                     ; for Device 2.
2C   DJNZ r3,  DELAY ; Short delay.
2E   CLR  P0
30   POP  IMR        ; Reactivate interrupts.
32   RET

  HANDLER:            ; Interrupt for Device 0.
33   LD   r2,  #00h  ; Reset counter in main loop.
35   CALL SEND
38   IRET            ; Interrupt Handler is done.
  .END
```

Figure 4.2. Example Program (part 2)

Each node in the call graph contains two pieces of information. The first is the value of the program counter, and the second is the value of the IMR. For this diagram, representation of the IMR has been simplified to two bits; the first represents the master mask bit, and the second represents the IRQ0 mask bit. (The example only makes use of interrupt zero.)

Control flow begins in the upper left corner of the graph, at the label "INIT". At this time, the program counter is 0C, and the IMR is cleared. Across the top of Figure 4.3, straight line initialization code is executed, with no interrupt enabled. At the node labeled "START", the PC has value 1C, and both the IRQ0 and master IMR bits have been set. From this point on, all nodes with an IMR of 11 have an outgoing edge leading to the interrupt handler.

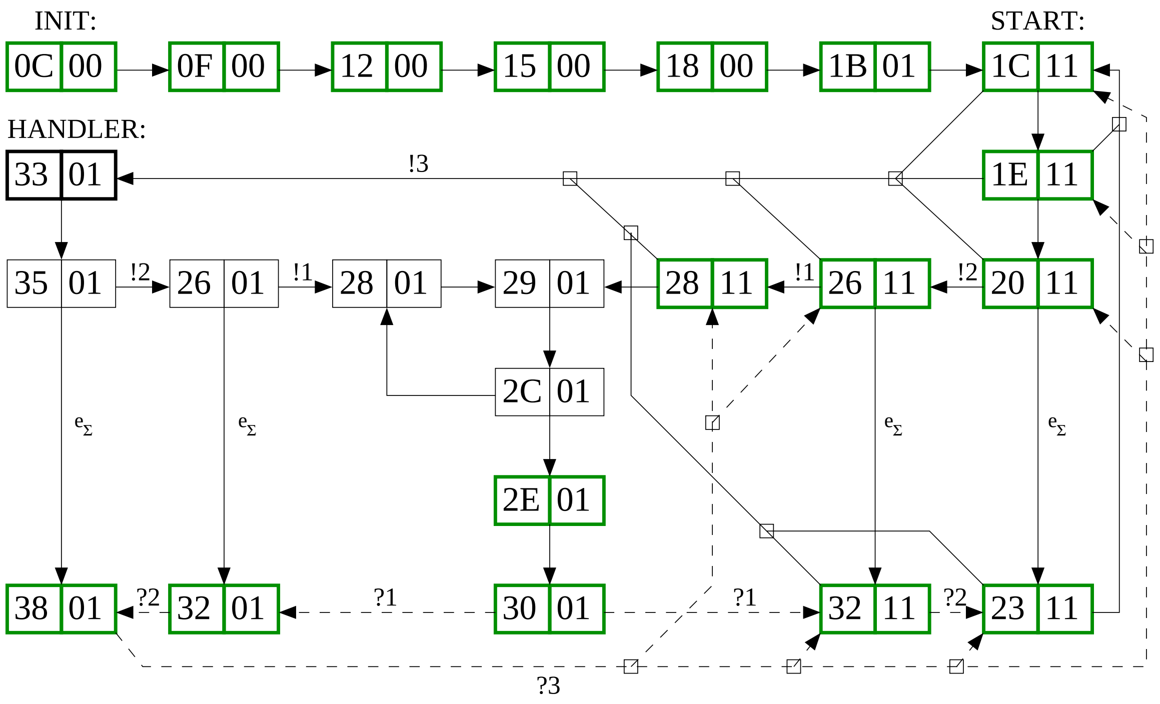


Figure 4.3. Example Program Flow Graph

Edges labeled with "!" or "?" correspond to pushing and popping operations, respectively. The number following the punctuation on these edges indicates the number of bytes involved in the stack operation. The PUSH instruction pushes one byte on the stack, while CALL pushes two, and an interrupt pushes three. Pop edges are distinguished with dashed lines. Additional "summary" edges generated by the analysis are labeled "e_Σ", and will be explained in a later section.

In order to calculate maximum possible stack size, a depth-first traversal of the graph is made, totaling up the push values of all the edges along each path. Pop edges are not traversed, but the summary edges are. In the figure, this means that the dashed edges are not considered during the search for the longest possible stack length. From this, a path with maximal stack size is found.

For the example program, the maximum stack size can be seen to be nine bytes. In short, the maximal path is to take an interrupt from node (28,11), where the size is already three. The interrupt pushes three more bytes on the stack to get to the handler, at (33,01). From there, the interrupt takes the edges to nodes (26,01) and (28,01), adding three more bytes to the stack for a grand total of nine bytes.

All of the bold-edged nodes in the flow graph have a finite worst-case path to reach the interrupt handler. Nodes with thin edges, however, defy the analysis presented in this chapter when trying to calculate maximum interrupt latency. Chapter 5 will present the modifications necessary for the deadline analysis algorithm to bound interrupt latency at these nodes.

4.3 Model Checking

4.3.1 The Z86 Assembly Language

As alluded to earlier, the Z86 architecture has several special registers that deal with interrupts. The Interrupt Mask Register (IMR) contains information about which interrupts are turned on. Six of the bits control interrupts zero through five. The Interrupt Request Register (IRQ) indicates which interrupts have fired, but have yet to be handled. A third register is used to set interrupt arrival tie-breaking priorities, but tie-breaking does not come into play for this analysis.

The Z86 architecture supports an indirect register addressing mode. The analysis relies on the unchecked assumption that the special registers IMR, IRQ, and SP are not altered indirectly. Checking the assumption would require further analysis of all 256 registers and is left to future work.

The analysis algorithms in this dissertation restrict direct manipulation the IMR, IRQ, and SP registers, as discussed below. Other forms of use can be located easily, and are explicitly flagged as errors by an early pass of the tool.

- IMR values are allowed to be pushed on the stack, popped from the stack, or manipulated by any binary operation in which one operand is a numeric constant, and the other is the IMR. While other operations on the IMR are certainly possible to express in the Z86 assembly language, the analyses presented here do not allow such operations. These constraints on the expressiveness of the language allow precise sets of possible IMR values to be calculated for all program points, and have proven to be sufficiently flexible to admit all seven of the commercial benchmarks.

- IRQ is read only. The Z86 architecture allows programs to write to the IRQ register, essentially raising interrupt requests manually. There does not appear to be an inherent barrier to analyzing programs that use this feature, but it was not encountered in any of the benchmark programs, so it has not been modeled in these analyses.

- SP is allowed to be manipulated implicitly by stack-specific instructions or explicitly by initialization instructions. In the commercial benchmarks, it is not unusual for the stack to be cleared by an explicit reloading of the initial stack pointer, so this is admitted by the analysis, and is noted by a special *nuke stack* edge in the control flow graphs. However, the analysis algorithms do not allow the stack pointer to be reinitialized to arbitrary values, and will reject any program that loads more than one numeric constant into the stack pointer register. The *nuke* edge is a special case, which for simplicity will be omitted from discussion for the rest of the chapter; in the stack-size analysis, it is treated like an e edge from the *start* node of the program to the destination of the *nuke* edge.

There are other unchecked assumptions in this dissertation's stack-size analysis. It is assumed that the system stack does not overlap with registers used for other purposes, and therefore is not corrupted by other instructions. The very purpose of this stack-size analysis is to help the system developer check this assumption.

It is also assumed that the Z86 watchdog timer functionality does not interfere with control flow. The Z86 has a WDT opcode, which once executed, will reset the processor if another WDT opcode is not executed within a programmable deadline. This feature is intended to allow system designers to prevent the software from locking up by entering an unintended infinite loop or other unforeseen control flow. Watchdog timer reset therefore signals a serious error in the program, and the analyses currently assume that watchdog timer effects do not occur. Checking this assumption is an interesting problem all by itself, one for which these analyses may be extended to tackle in future work.

4.3.2 From Z86 Assembly Code to a Flow Graph

Given a Z86 assembly program, a CFG is constructed in which each vertex is labeled with a PC value and an IMR value. The start vertex is labeled with 1) the PC value for the first line of the program, and 2) the IMR value where all bits are 0. The graph is built in a demand-driven way such that only nodes that are reachable from the start node are explored. Each edge represents a possible step of computation. The flow graph is a conservative representation of the program: while each possible computation at the program level is represented as a path in the graph, there may be paths that do not correspond to a computation. (This is the False Path problem, as mentioned earlier in Sections 2.2 and 3.4.3.)

There are ten kinds of edges, each with a distinctive label, as shown in Figure 4.4. An edge label indicates how many elements are placed on the stack (or removed from the stack) by the corresponding step of computation. An edge with label "e" or "e_Σ" has weight 0, an edge with label "$!n \ldots$" has weight n, and an edge with label "$?n \ldots$" has weight $-n$. Label "unk" is used as an abbreviation of "unknown" in connection with edges of weight 1 that are unrelated to IMR. Some of the labels also contain the actual values placed on the stack. Many instructions do not change the stack; they are represented rather anonymously with an edge labeled "e", which stands for an "epsilon transition" in the equivalent automaton. Two kinds of edges do not correspond to any instruction: the edges for implicit interrupt calls, and the summary edges, "e_Σ", which are a special class of the epsilon edges.

Conceptually, the graph is built in three passes. First, the edges for the normal, non-interrupt code are inserted. This includes all instructions that place values on

instruction format	edge label	computation step
⟨various⟩	e	Epsilon edge – no stack change.
⟨summary⟩	e_Σ	Epsilon summary edge – no stack change.
PUSH IMR	!1	the value of the IMR is placed on the stack.
PUSH ⟨not IMR⟩	!1	some value (not IMR) is placed on the stack.
CALL ⟨label⟩	!2	procedure call. (return address saved)
⟨interrupt call⟩	!3	implicit interrupt call. (return + flags saved)
POP IMR	?1	the IMR is assigned the value on top of stack.
POP ⟨not IMR⟩	?1	some register (not the IMR) is assigned the value on top of the stack.
RET	?2	return from procedure call.
IRET	?3	return from an interrupt handler.

Figure 4.4. Instructions and the corresponding edge labels

$$\begin{tikzcd} m \arrow[r, "e"] & p \text{ POP IMR} \arrow[d, dashed, "?1"] \\ n \arrow[u, "\text{!1 (IMR)}"] \arrow[r, dashed, "e_\Sigma"] & q \end{tikzcd}$$

$$\begin{tikzcd} m \arrow[r, "e"] & p \text{ POP } \langle\text{not IMR}\rangle \arrow[d, dashed, "\text{?1 ``unk''}"] \\ n \arrow[u, "\text{!1 ``unk''}"] \arrow[r, dashed, "e_\Sigma"] & q \end{tikzcd}$$

$$\begin{tikzcd} m \arrow[r, "e"] & p \text{ RET} \arrow[d, dashed, "?2"] \\ n \arrow[u, "\text{!2 (a)}"] \arrow[r, dashed, "e_\Sigma"] & q \end{tikzcd}$$

$$\begin{tikzcd} m \arrow[r, "e"] & p \text{ IRET} \arrow[d, dashed, "!3"] \\ n \arrow[u, "\text{!3 (s,r)}"] \arrow[r, dashed, "e_\Sigma"] & q \end{tikzcd}$$

Figure 4.5. Rules for Inserting Summary Edges

the stack, or do not change the stack; instructions that pop values from the stack are not yet considered. Second, implicit interrupt call edges are inserted from all program points, based upon the set of possible IMR values already known from the first pass. Finally, the graph is closed under the four rules shown in Figure 4.5 and the rule that the epsilon edges, (labeled "e" or "e_Σ",) form a transitive relation. In each of the four rules, the intention is that if the solid edges are present, then the dashed edges must also be present.

The four rules illustrated in Figure 4.5 govern the generation of 1) *pop edges* that correspond to removing values from the stack, and 2) *epsilon summary edges* with label "e_Σ" that connect the point where values are placed on the stack to the point where the same values are removed.

Pop edges are not used in this chapter's stack-size analysis, but matter in later chapters. The e_Σ edges summarize a net stack size change of zero across a segment of code with both push and pop edges.

For example, consider in detail the first rule in the upper left of Figure 4.5. The node n is for an instruction "PUSH IMR", and there is an edge from n to m that models the IMR being placed on the stack. Moreover, there is an edge labeled "e" from the node m to a node p. The node p is for an instruction "POP IMR". There could be an arbitrary number of instructions between m and p with a net stack change of zero, but because epsilon edges are transitive, these cases are the same as the single edge case. It is now straightforward to calculate the label of a node q that will be the target of an edge (a pop edge) from p. The pop edge represents removing the IMR value from the stack and assigning it to the IMR register. The epsilon summary edge, labeled "e_Σ", is inserted from n to q. The epsilon summary edge reflects that the stack size is the same at n and q, so it is warranted to allow a shortcut.

Notice that there can be more than one outgoing edge from a node for an instruction that removes elements from the stack.

The stack size analysis algorithm can be understood as a demand-driven version of an algorithm for model checking of pushdown systems [79]. Unlike [79], this algorithm generates pop edges on demand, thereby ensuring that only reachable nodes are considered. The closure process can be done in $O(n^3)$ time where n is the number of nodes in the final flow graph [53].

4.3.3 Stack-Size Analysis

To calculate a stack-size estimate, it is sufficient to consider only edges with weights 0 or higher. This is a fundamental property of all graphs that have been closed in the sense explained earlier in Section 3.2.2. The analysis can now calculate a stack-size estimate by a straightforward depth-first traversal. For all paths from the start node of the graph, the traversal calculates the sum of the weights of the edges on the path. The maximum number found in this way is the estimated stack size. In case the traversal encounters a loop with at least one edge of weight 1 or more, then the stack-size estimate is "infinite." Such a loop indicates a possibly infinite loop in the program where the stack grows each time around the loop. Such a situation may signify a programming error.

4.3.4 Type Checking of Stack Elements

The goal of the type check is to ensure that various instructions are executed in a machine state where the top of the stack is of the expected type. The type-checking algorithm uses an implicit type system with just four types:

$$\text{type} \ ::= \ !1 \text{ “IMR”} \ | \ !1 \text{ “unk”} \ | \ !2 \ | \ !3.$$

Edge labels can be mapped to types in the obvious way.

The type check ensures that for every path of the form

$$n \xrightarrow{!\ldots} m \overset{\text{e}}{\rightarrow} p$$

where p models one of “POP IMR”, “POP ⟨not IMR⟩”, “RET”, “IRET”, we have one of the four situations

$$\begin{array}{ll}
n \xrightarrow{!1\ (\text{IMR})} m \overset{\text{e}}{\rightarrow} p & \text{and } p \text{ models “POP IMR”} \\
n \xrightarrow{!1\ \text{“unk”}} m \overset{\text{e}}{\rightarrow} p & \text{and } p \text{ models “POP } \langle\text{not IMR}\rangle\text{”} \\
n \xrightarrow{!2\ (\text{a})} m \overset{\text{e}}{\rightarrow} p & \text{and } p \text{ models “RET”} \\
n \xrightarrow{!3\ (\text{r,a})} m \overset{\text{e}}{\rightarrow} p & \text{and } p \text{ models “IRET”.}
\end{array}$$

Such checks correspond to the safety checks of Palsberg and Schwartzbach [74, 76], and can be implemented efficiently as outlined in Section 3.2.2.

4.4 Experimental Results

4.4.1 Benchmarks

The seven proprietary microcontroller systems used for these experiments are provided by Greenhill Manufacturing, Ltd. (http://www.greenhillmfg.com/). Three of the controllers, “ZTurk”, “GTurk” and “CTurk”, drive multiple-zone evaporative cooling systems, often present in poultry barns, particularly for turkeys. “Fan” and “Serial” run variable speed cooling fans for forced ventilation structures, such as modern swine barns. “Rop” and “DRop” handle a water quality / reverse-osmosis filtering system commonly used in car washes.

In addition to the commercial systems, test results are included for a smaller test program written to display more interesting interrupt behavior than the commercial benchmarks. This benchmark is labeled “Micro00”, and its full text can be found in Appendix A.

4.4.2 Infrastructure

The Zilog Architecture Resource-Bounding Infrastructure includes an instruction cycle-level simulator for the Z86C30 architecture, in order to more closely examine the execution of programs. The specifications for the simulator are taken from the Zilog product specification available for this architecture, [100]. Where the specifications have been found to be ambiguous, worst-case assumptions have been made. Simulation has been chosen because the actual Z86 chips do not contain hardware provisions for profiling, and because running software on the Z86C30 requires permanently burning a particular program into a "one-time programmable" chip, which would quickly become cost-prohibitive in a research setting. The commercially available development emulator for this architecture has very limited support for timing analysis, and does not allow single-step examination of interrupt behavior.

All of the microcontroller systems available to us from Greenhill have the Z86 processors built into a circuit board with several other peripheral chips, and the software for the systems reflects this fact. The simulator must therefore include models of this external hardware in order to properly simulate the environment of the program. Simple state machines provide the minimal interaction necessary to simulate the normal execution paths of the systems. These state machine models are generally constructed from the hardware manufacturers' specifications for the individual components, and assume worst-case delays wherever possible.

It appears to be a fundamental property of the examined embedded systems that off-chip resources must be considered in order to undertake any comprehensive modeling of the system. While this kind of information should be readily available to the system designer in a production environment, it means that tools like the prototype presented here are less likely to be able to be applied to new systems "out of the box."

ZARBI includes pilot scripts that drive the simulator using a genetic algorithm to search for interrupt conditions that lead to large stack heights. (See Section 6.2.4 for details.) Because these conditions yield actual executable paths in the software, (rather than "false paths",) they provide realistic lower bounds for maximal stack height, against which static analysis results can be compared.

4.4.3 Building the graph

This section displays results taken from running the stack-checking algorithm on the test suite of programs.

All algorithms presented in this chapter are implemented in Java, and run on the IBM Java2 SDK 1.3. Runs were made on a 500 MHz Pentium3-based laptop.

The stack-checking implementation has been optimized for speed, but avenues for further optimization remain. Space usage has not been optimized, and could be reduced significantly with further effort. However, the current prototype implementation is sufficiently fast (most runs take a few seconds) and sufficiently compact (at

Building the graph				
Program	Nodes	Edges	Time	Space
CTurk	1,209	2,316	4.01 s	31.6 MB
GTurk	1,581	3,101	4.20 s	32.2 MB
ZTurk	1,493	2,885	4.12 s	32.1 MB
DRop	1,138	2,043	4.02 s	31.1 MB
Rop	1,217	2,278	4.08 s	31.7 MB
Fan	5,149	17,195	5.13 s	39.3 MB
Serial	394	1,082	3.78 s	31.0 MB
Micro00	148	222	3.16 s	34.9 MB

Figure 4.6. Graph size and resource usage for benchmarks

most 40 MB) for experimentation. Naturally, both speed and space usage could be improved if implemented in C.

All time measurements are averages over 10 runs. To prevent external factors such as hard disk speed or cache behavior from influencing the simulator results, several "warm-up" runs are made prior to the recorded runs. The reported time usage is the real time elapsed for the run from start to finish.

The space measurements were made with `top`. The space reported is the maximum total size during the run, including space taken by the Java virtual machine, garbage collector, and JIT. Measured space usage was deterministic (the same for each run of the same program).

Roughly half of the time and space usage reported in Figure 4.6 is spent building the graph; the rest is spent starting the Java virtual machine and parsing the Z86 assembler file. The parser uses the tools `JavaCC` [96] and `JTB` [91] for parser generation and syntax tree manipulation.

4.4.4 Stack-Size Analysis

The upper bounds on the stack sizes found by the analysis are reported in Figure 4.7, in the column labeled "Upper Bound". The lower bounds reported in Figure 4.7 are from the genetic algorithm search with the simulator; because these represent stack heights from known execution traces, the true maximum stack height must be no less than these lower bounds.

The stack-size analysis typically takes around 0.1 seconds, and takes little extra memory beyond the base size of the Java virtual machine. Note that the columns

Stack-size analysis				
Program	Lower Bound	Upper Bound	Total Time	Total Space
CTurk	17	18	4.11 s	31.6 MB
GTurk	16	17	4.31 s	32.2 MB
ZTurk	16	17	4.22 s	32.1 MB
DRop	12	14	4.14 s	31.1 MB
Rop	12	14	4.18 s	31.8 MB
Fan	11	N/A	N/A	N/A
Serial	10	10	3.87 s	31.0 MB
Micro00	37	37	3.21 s	34.9 MB

Figure 4.7. Stack size results

Total Time and Total Space include the cost of building the graph, as well as the stack size analysis.

The analysis presented here is unable to ascertain an upper bound on the program "Fan" because it has the assembler equivalent of a `for` loop with a PUSH in the body. This corresponds to a positive cycle in the CFG (see Section 3.2.3). While it is obvious to a programmer that the number of loop iterations (and therefore the stack size) is bounded for this particular loop, the analysis algorithm cannot see the bound based solely on the PC and IMR registers. The prototype implementation includes provisions in its data structures to model this kind of control flow, but the analysis extensions have not been implemented at this time.

Despite efforts to limit unrealizable control flow paths in the graphs, the upper bounds presented in Figure 4.7 may not correspond to genuine execution paths in the running microcontroller programs. The following approach is used to evaluate the precision of the upper bounds by finding lower bounds in actual program runs.

ZARBI's cycle-level simulator for the Z86E30 architecture includes all but a few obscure processor features that are not used by the benchmark programs. The simulator can interact with state machine models of external devices, including an intelligent LCD display, an 8-bit Analog-to-Digital converter, a 9600-baud RS-485 serial port, and a 64-byte EEPROM chip. The simulator can monitor stack size, and records the maximal value together with the corresponding program path. Any run of one of the benchmark programs with some interrupt schedule will generate a lower bound on the stack height; a genetic algorithm directs the evolution of interrupt schedules to search for a tight lower bound.

The input to the simulator is an assembly program and an interrupt schedule. The schedule consists of a number of interrupt request sequences that should be fired during the run in order to test the assembly program. The format of the interrupt schedules supports both single-point interrupts and periodic interrupts. A full description of the interrupt schedule file format can be found in Appendix C.

For completeness, the experiments presented above used several strategies to search for an interrupt schedule that gave as tight a lower bound as possible. These strategies included simulation with 1) an "expert" interrupt schedule written by a person familiar with the Greenhill microcontroller systems, 2) 1,000 randomized schedules, and 3) 1,000 schedules generated by the genetic algorithm. The genetic algorithm consistently matched or outperformed the results of the other two approaches. The lower bounds found by the simulator with the winning interrupt schedule are reported in Figure 4.7.

4.4.5 Type Checking of Stack Elements

For the six benchmark programs for which the analysis produced a finite stack size, all stack operations type check. This was also true for all additional test inputs that were not written with deliberate stack manipulation errors. All of the test inputs with intentionally broken stack operations were detected and properly flagged. The tool carries out the checks while executing the closure rules that insert pop edges.

The algorithm for type checking does not apply to programs with unbounded stack size. Intuitively, this is because in such programs, it is not possible to match the push and pop operations "one to one."

4.5 Summary

The experiments shown in this chapter were designed to explore the question, "Can modeling just the program counter and interrupt mask registers lead to a useful programming tool?" The answer is certainly yes.

The stack size checking algorithm was able to provide tight upper bounds on six of the seven proprietary programs. Furthermore, it effectively type checked the stack operations on those six programs.

The seventh program defies analysis only because of a single loop which depends on other registers to determine stack size. While this kind of limitation is symptomatic of the undecidability of this problem in the general case, much work has been done in the past on handling simple instances, as are likely to occur in assembly programs of this type. Identification of induction variables and loop unrolling [62], and loop-invariant specification [67, 68] are successful techniques that may be combined with the analyses presented here to tackle the upper bounds on the seventh program.

As for calculating maximum interrupt latency, PC and IMR values alone are not sufficiently precise to differentiate nodes with disparate latencies; latency analysis will be covered in depth in Chapter 5.

This work has the potential to impact far more assembly languages than that of the Z86. The maskable, vectored interrupt architecture present on the Z86 is very similar to many other processors, such as the Motorola 68000 family, and many RISC DSP chips. Palm Pilots, handheld digital phones, and many other interrupt-oriented applications use software that could be amenable to analysis along the lines presented in this dissertation. While the Z86 programs examined here are on the order of 4K in size, average Palm Pilot programs are 100K in size, with about three times as many interrupt vectors. Estimating based upon current results, this would result in graphs with a few hundred thousand nodes, and a few million edges – still within grasp of current machine power for analysis. The larger instruction sets and register sets of these processors are a largely orthogonal issue to the complexity of the analysis, and only add details to the complexity of the implementation.

A key difference between the Z86 and larger interrupt-oriented processors is the issue of program progress. With code in ROM, and no capacity for bus errors, the Z86 processor is guaranteed to always proceed in its computation, regardless of what garbage instructions it might be forced to execute. (It is possible for a poorly-written Z86 program to jump to data constants stored in ROM, which would result in "garbage" being executed.) In short, at least one of the edges leaving each node in the graph is guaranteed to be taken upon execution. Not so with more complex processors, where a badly formed jump address could cause computation to stop, due to a bus error, a protection error, or a misaligned memory address. For these reasons, additional safeguards, like Typed Assembly Language [59] would be required in order to provide the necessary structure to guarantee program progress in such a scaled-up framework. As an added bonus, such typing annotations may assist in eliminating "yellow" latency ambiguity in the graph, as will be explained in Chapter 5, by providing much-needed limits on the flow of critical data. Finally, type systems could enforce the safety checks on indirect addressing modes and direct addressing instructions that the current implementation neglects.

The stack size estimation technique presented in this chapter is one of the first to give an efficient and useful static analysis of assembly code. It employs static analysis to provide safe, tight bounds on stack size for interrupt-driven Z86 microcontroller systems.

The next chapter will present techniques for bounding interrupt latency in interrupt-driven systems.

5 DEADLINE ANALYSIS

The deadline analysis algorithm presented in this chapter combines timing oracles with static analysis to provide safe bounds on interrupt latency for real-time systems implemented on the Z86 platform.

This chapter presents the difficulties of deadline analysis in such systems, the algorithm used for deadline analysis in this dissertation, and the results of applying the prototype implementation to a suite of commercial embedded systems.

After a brief overview in Section 5.1, Section 5.2 presents a program which will be used as a running example throughout rest of the chapter. Section 5.2.3 presents the concept of oracles, and Section 5.2.4 presents multi-resolution static analysis. In Section 5.3, experimental results are given, and Section 5.4 walks through an interactive deadline-analysis session with ZARBI.

5.1 Overview

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 many static-checking tools available, including type checkers [17], bytecode verifiers [49], and model checkers [19], as well as numerous tools for supporting the test process. Verification of timing properties is more difficult, but progress has been made toward understanding the foundations of checking the timing properties of real-time software in work such as [5] and [6]. Major open issues still remain, due to the low-level nature of real-time systems. Many are 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 [99] is not concerned with timing properties.

5.1.1 The Deadline Analysis Problem

The analysis presented later in this chapter checks timing properties of real-time assembly code. A prototype tool has been constructed as a demonstration of the practical benefits of these techniques. 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 the systems analyzed in this dissertation can 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, for example, the system has just disabled interrupt handling and will be doing other work for the next two million clock cycles. Deadline analysis seeks 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, by trying a suite of interrupt schedules and measuring 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 a combinatorially explosive problem. 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. As shown in Chapter 4, static analysis can be successfully employed to bound stack usage in interrupt-driven systems. However, when timing analysis was applied to the model presented in Chapter 4, worst-case execution time could not be estimated for most of the paths in the program.

Static timing analysis for embedded systems cannot succeed without information about the behavior of external devices that interface with the embedded processor. For example, if the processor 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 that there is 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 presented in Chapter 4 fails to perform useful deadline analysis.

This chapter explores the thesis that better results can be obtained by *combining* 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 [37]. In the deadline analysis setting, static analysis can reduce the required testing effort, allowing the testing effort to be more *focused* on key areas of the code that affect deadlines.

The deadline analysis presented here uses test oracles [85] to answer certain worst-case execution time (WCET) questions that cannot possibly or easily be answered by static analysis. An oracle asserts to the static analysis that if execution reaches program point A, then it will reach program point B at most t microseconds later. Returning briefly to the high-level CFG abstractions of Chapter 3, oracle assertions are expressed in the CFG's as *time summary edges* (Section 3.5.1). When A and B are close, then a much smaller testing effort is required to verify such an oracle assertion than to do the entire deadline analysis. Moreover, if more than one oracle assertion is needed for a program, the work of validating each assertion can be done in parallel. The 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 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 [10, 20, 27, 30, 77, 93, 97]. 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.

5.1.2 Results

ZARBI has been designed and implemented to be used as a tool for integrated deadline and WCET analysis of interrupt-driven assembly code. Expressed in simplest terms, the ZARBI methodology is:

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

For six commercial microcontroller programs, each on the order of 1000 lines of code, less than 17 oracles were sufficient to complete deadline analysis. In the

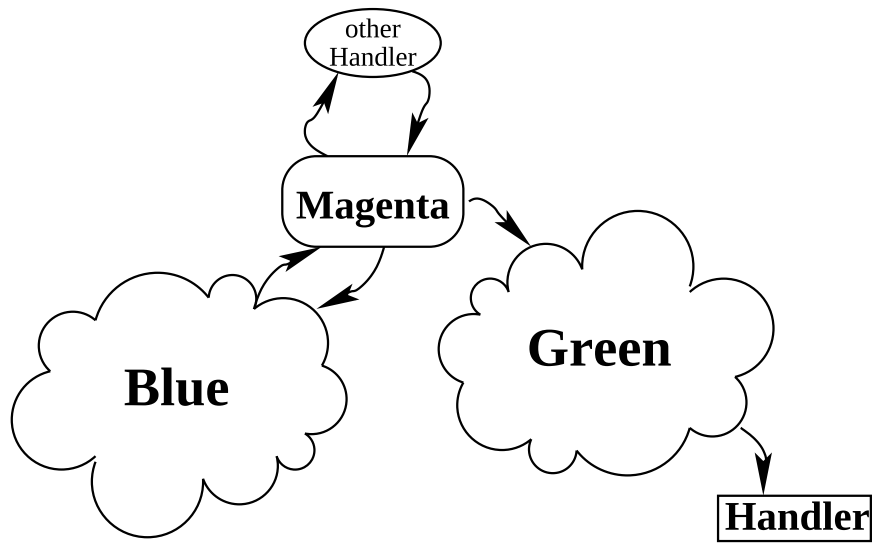


Figure 5.1. Coloring a Flow Graph

experimental session presented in Section 5.4, an expert user was able to interactively add all of the required oracles for one of the commercial benchmarks in less than an hour.

The technique presented here uses a multi-resolution analysis (Section 3.4.4), which allows exploration of difficult segments of the control flow graph in sufficient depth to bound the latency while avoiding the intractable complexity that would arise from using such fine-grained analysis over the whole program.

The 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 WCET can be estimated, *Magenta* means that starvation is possible, *Blue* means that starvation is possible at a later node, *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 the test suite, no red nodes were found, the analysis was able to eliminate all yellow nodes with the addition of oracles, and very few nodes were magenta.

Figure 5.1 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”.

The deadline analysis presented here 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.

5.2 Example Analysis

5.2.1 A Program and its Flow Graph

The example program shown in Figure 5.2 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. These addresses will be used throughout the rest of this section to refer to specific lines of the example.

Figure 5.3 shows the flow graph constructed for the example program in Figure 5.2. 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 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.

- 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 shown 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,

```
     .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 PROGAM
23   JP   LOOP           ; An infinite loop.
                         ;SUBROUTINES
  PROC:                  ; This subroutine just pushes and 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 5.2. Example Program

the Z86 pushes two bytes of return address and one byte of condition code bits onto

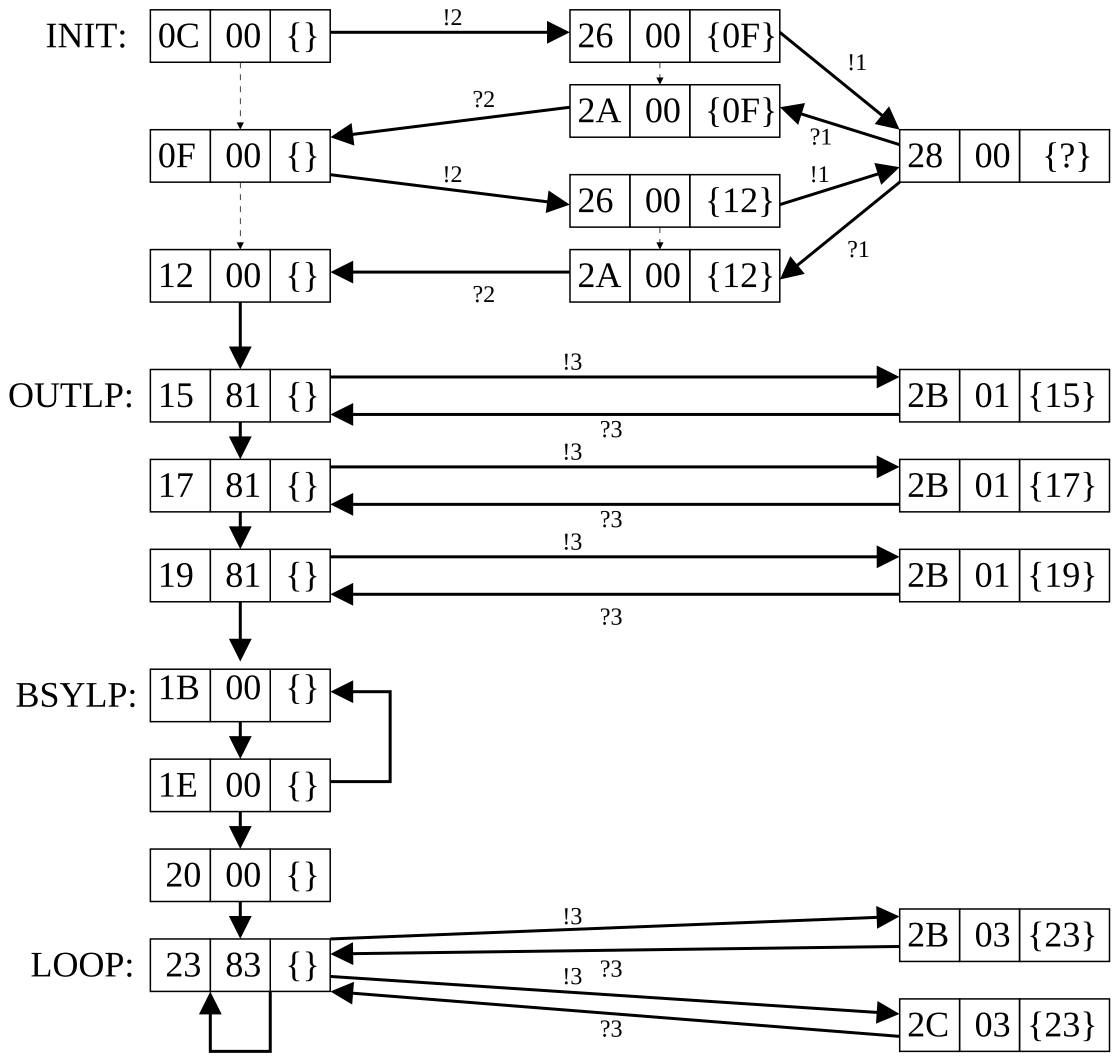


Figure 5.3. Example Program Flow Graph

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 earlier in Chapter 4.

5.2.2 Initial Coloring of the Example Graph

The designer of the example program in Figure 5.2 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, the algorithm 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 5.2.4.

- *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 3.5).

- *Red* nodes are those from which it is impossible to reach the handler of interest. In ZARBI's 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 in this chapter.

- *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 5.3) 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 5.3 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 [16]) 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.

5.2.3 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 found in several of the 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, 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

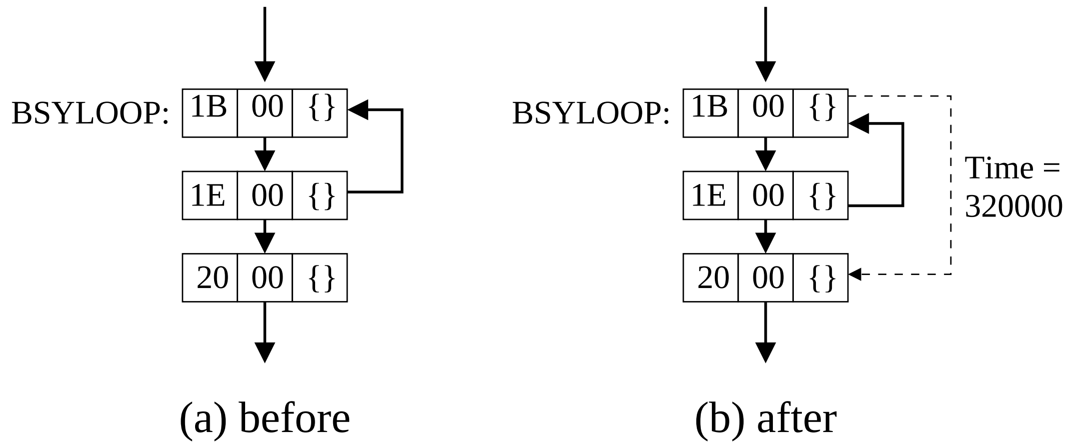


Figure 5.4. Time Summary Oracle in the Example

$Address_1$ 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 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.

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 5.4. 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.

This dissertation uses 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 [27,62]. 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.

The correctness of assertions made by the user to the oracle are taken for granted by the current system. Assertions must be admissible (Section 3.5.1) for the overall analysis to produce correct results. 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 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.

5.2.4 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 node.) However, the imprecision of having nodes distinguished by only one element of stack context (analogous to 1-CFA in flow analysis parlance [90]), 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 genuine 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, (see Section 2.2); 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. The CFG is constructed using *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}], as shown in Figure 5.5. This approach is inspired by Plevyak and Chien [78]. Independently of our work, Guyer and Lin [36] have also used multi-resolution analysis.

Multi-resolution analysis takes place automatically; the algorithm (shown in Section 6.3.2) iteratively identifies nodes that are both *border yellow* and stack popping

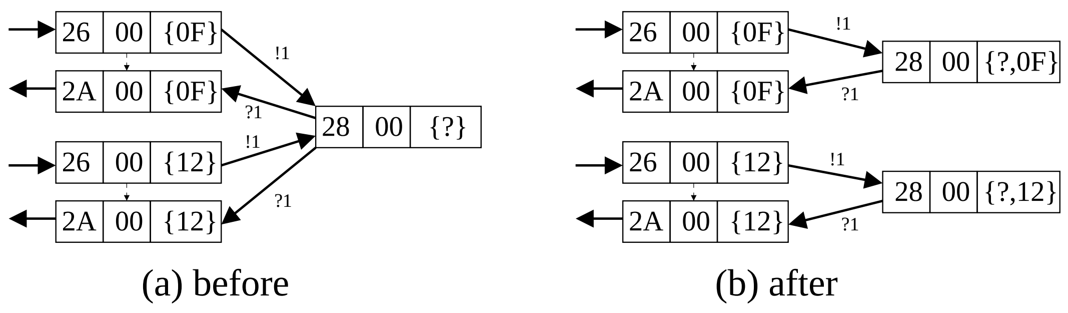


Figure 5.5. Example Program Adaptive Slicing

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 explained 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, which is verified by a previous phase of the tool, (see Chapter 4.) The full details of the adaptive slicing can be found in Chapter 6.

5.2.5 Magenta and Blue Nodes

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 (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 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.)

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 and is beyond the scope of this dissertation. 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.

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 5.1.

The algorithm for coloring the graph is summarized in Computation Tree Logic [24] notation in Figure 5.6. H is a predicate that is true for a node when that node is the first instruction of the interrupt handler of interest. In CTL notation, AF means "exists globally", which can be thought of as "inevitable". So $Green \equiv AF(UltraGreen)$ means that $Green$ nodes are those for which all outgoing edges inevitably reach $UltraGreen$ nodes. Notation EF means "exists eventually", or "reachable". EX means that there is an outgoing edge that leads immediately to the predicate. Thus, $Magenta \equiv EF(Green) \ \wedge \ EX(handler \notin H)$ says that a $Magenta$ node has a path that eventually reaches $Green$, and a path that leads

$UltraGreen$	$\equiv$	H	$\equiv$	Head of handler of interest.
$Green$	$\equiv$	$AF(UltraGreen)$	$\equiv$	Inevitable that computation will reach an UltraGreen node.
$Magenta$	$\equiv$	$EF(Green)$ $\wedge\ EX(handler \notin H)$	$\equiv$	Path exists to Green, and to non-Green IRQ handler.
$Blue$	$\equiv$	$AF(Magenta)$	$\equiv$	Inevitable that computation will reach a Magenta node.
Red	$\equiv$	$\neg EF(UltraGreen)$	$\equiv$	Not possible to reach an UltraGreen node.
$Yellow$	$\equiv$	$\neg(Red \vee Green$ $\vee Magenta \vee Blue)$	$\equiv$	Don't Know.

Figure 5.6. Coloring Graph for Latency Analysis

in one edge to a non-*Green* interrupt handler. The ZARBI implementation of this coloring algorithm is explored in Chapter 6.

Returning to the control flow graph from Figure 5.3, 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 [100]. 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.

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 Chapter 4
ICSE01	55	1	Example from Chapter 4
FSE03	35	2	Example from Chapter 5

Figure 5.7. Benchmark Characteristics

5.3 Experimental Results

The following sections present experiments applying the prototype implementation of this analysis to the suite of commercially available microcontroller systems. Following these results, Section 5.4 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.

5.3.1 Benchmark Characteristics

The benchmarks used for evaluating the deadline analysis (Figure 5.7) are the same suite of test inputs used in Chapter 4 with the addition of the examples from Figure 4.1 ("ICSE01") and Figure 5.2 ("FSE03"). The commercial program "Fan" has been omitted because the stack size analysis presented in the previous chapter cannot 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.

Each of the commercial systems underwent months of testing prior to actual production, but an overall deadline analysis of the systems was not performed because no such tools could be found.

5.3.2 Measurements

The results shown in Figure 5.8 give the final percentages of nodes by color after completion of the deadline analysis algorithm. For clarity of presentation, interrupt

	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 5.8. Results With Completed Oracles

sources in the tables are numbered 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 5.8 may not total precisely to 100%.

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 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

		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 5.9. Adaptive Slicing vs. Fixed k-CFA

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 5.9 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 5.4,) anything that causes larger graphs potentially creates more yellow nodes, adding more data to the output of the tool, and making the entire process increasingly difficult.

Figure 5.10 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.

	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 5.10. Oracle Information Provided

5.3.3 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 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 the total uninterrupted WCET of the magenta cloud could be less than the minimum period of the interfering interrupt handler(s), which would make it possible to reason about these sections with a first-order*worst-case response time analysis* [94] 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.

5.4 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`.

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 Appendix E.

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 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 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 Chapter 6.

5.5 Summary

The algorithms presented in this chapter perform deadline analysis on interrupt-driven assembly code. Static analysis was able to reduce the required testing effort to concentrate on the validity of certain oracle assertions about timing.

In 30% of the analyses of a particular interrupt handler for a particular benchmark, the deadline analysis was able to firmly bound maximum interrupt latency. In the remaining 60% of the cases, the analysis reduced the size of the testing problem by an average of 98%. While the testing of the oracles and remaining magenta nodes is still a large task, it is several orders of magnitude smaller than the testing problem without the deadline analysis presented in this chapter.

The multi-resolution analysis allows for compact and efficient representation of timing properties while smoothly incorporating the oracles. For each of the test inputs, less than 17 oracles are sufficient, and these can be added in an interactive fashion until the deadline analysis is complete. In the experiments, it was observed that 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. (This does not include the subsequent exhaustive testing of the oracles, which would normally be done even without any analysis by ZARBI.)

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 [60, 61], the Intel 8051 family [45], the National Semiconductor COPS8 family [64], as well as several RISC DSP architectures, and other special purpose chips.

This chapter presents one of the first algorithms to allow deadline analysis of interrupt-driven assembly code. The proof-of-concept implementation demonstrates its usefulness when run on commercial-grade real-time software. ZARBI is also one of the first tools to incorporate static analysis with testing oracles in an interactive fashion.

The next chapter presents fine-grained details of the tools demonstrated above, including implementation issues, limitations, and features intended for future use.

6 ZILOG ARCHITECTURE RESOURCE-BOUNDING INFRASTRUCTURE

The previous chapters have presented algorithms for stack size analysis and deadline analysis at a high level of abstraction without focusing on implementation details. This chapter is the compliment to that high level view, detailing the prototype tools that implement the resource bounding analyses described earlier.

6.1 Data Structures

The primary data structures used in ZARBI are for storing and manipulating the control flow graph representation. Four main classes are responsible for this function: GraphNode, GraphEdge, GraphID and GraphNexus.

The GraphNode is the central data structure and represents a single node in the control flow graph. Each GraphNode has a unique GraphID which contains the PC, IMR, and stack context for the GraphNode. The GraphID class exists to separate the methods for storing, manipulating, and comparing this information from the methods for graph building and traversals. GraphNodes have two arrays of GraphEdges: one for incoming edges and one for outgoing edges. The GraphNode class implements the GraphNodeInterface and can be used in the same graphs with other subclasses of the GraphNodeInterface interface.

The GraphEdge class implements the GraphEdgeInterface and represents a directed edge in the graph. GraphEdge has a reference to a source GraphNodeInterface and a destination GraphNodeInterface.

The GraphNexus structure is a joining point for all of the GraphNodes that have the same PC value. There is one GraphNexus for each line of code in the original Z86 program, plus several special nexi for other lines in the original assembler file. The GraphNexus serves as a bookkeeping entity, tracking data and statistics that all of its GraphNode *children* have in common. The most common searches performed on the graph during construction require finding a reference to particular GraphNode given only a PC and an IMR value. In this way, the array of GraphNexus objects is the backbone of the control flow graph, providing an organizational structure that is reflected in many of the ZARBI visualization and analysis tools.

While the GraphID class is primarily concerned with the PC, IMR, and stack values of GraphNodes, it also contains *flow source pointers*, references to the GraphID belonging to the node which pushed the current top element on the stack. This extra piece of information allows optimized construction of stack summary edges, (Sections 4.3.2 and 3.1), because each node already has a reference to the source node of the potential summary edge without executing an expensive search. The flow source pointer also allows comparisons of GraphID's to differentiate between identical top stack elements which were assigned by different source nodes. This technique causes top stack elements from particular source nodes to be treated as unique identifiers, which allows the stack size analysis to succeed in bounding the stack for certain degenerate cases where spurious summary edges would otherwise be constructed.

An abstract GraphTraversal class exists which allows both forward and backward breadth-first traversals of the control flow graphs. At least seven concrete subclasses of GraphTraversal exist, allowing many of the analysis passes in ZARBI to use a uniform traversal mechanism.

6.2 Stack Size Checking Tools

The next four subsections describe the implementation of four major parts of the stack size analysis presented in Chapter 4. The simplifier (Section 6.2.1) is a Z86 assembly language parser, which partially compiles the input programs and performs error checking that need not be repeated in later phases of analysis. The simulator (Section 6.2.2), state machine models (Section 6.2.3), and genetic search algorithm (Section 6.2.4) were all key components for finding the realistic lower bounds on maximum stack height presented in Chapter 4.

6.2.1 Simplifier

In order to avoid duplication of code and work in many of the tools in the ZARBI suite, Z86 programs are first passed through a simplification stage – essentially partially compiled – before being parsed in by later tools in the chain.

The simplifier expands all symbol table references to their final immediate values, labels each line of executable code, and performs a variety of error checks before passing the program on to the simulator, stack analysis, or deadline analysis engines.

The code for parsing raw Z86 assembly language files and building abstract syntax trees was largely automatically generated with the Java Tree Builder [91], another tool built at Purdue. Transformation of the abstract syntax tree into the simplified tree is handled through extensive use of the "Visitor" design pattern [33] and the Generic Java extension [13].

Error checking undertaken by the simplifier includes: ensuring that all arithmetic constants are in range to be stored in the available register size, whether those constants are expressed in binary, decimal, octal, or hexadecimal notation; checking that all jumps and calls are to valid code addresses; and identifying any unresolvable symbol references.

The simplifier outputs the partially compiled code into a file, which must then be parsed back in by the stricter grammar of later tools in the chain. This keeps compilation concerns like symbol table resolution completely separate from other analyses in ZARBI, and alleviates the need for redundant error checking in other tool components.

The output of the simplifier conforms to the grammar found in Appendix B.

6.2.2 Simulator

The ZARBI toolset includes a cycle-level simulator of the Z86E30 processor, constructed from Zilog's specifications [100]. Building a simulator based upon published specifications from the manufacturer can be error-prone, as such documents can be vague, incomplete, or simply wrong [26]. In the many cases where specifications were vague, the simulator was implemented with worst-case assumptions about the actual hardware. Nevertheless, the ZARBI simulator has not been validated against real Z86E30 chips in any way, and this would be an absolutely necessary step for an industrial strength version of ZARBI. The current simulator was intended primarily as an exploratory tool for evaluating the role of such a simulator in resource-bounding analyses. This simulator also does not implement several features of the Z86E30 architecture which are not used by the benchmark suite.

The simulator is approximately 7400 lines of Java code, not including the file created by the parser-generator tool. Both a graphical user interface (pictured in Figure 6.1,) and a command-line batch mode are available.

Upon graphical startup, the simulator reads in a simplified Z86 program and displays the ROM, register and flag windows. Single-step and break-point execution are available, with the various windows updating all register and flag values as the program is stepped through.

The simulator interface allows the user to alter values in any Z86 register and accurately displays processor state not otherwise available, even to the program running on raw hardware. (For example, exact timer count values.)

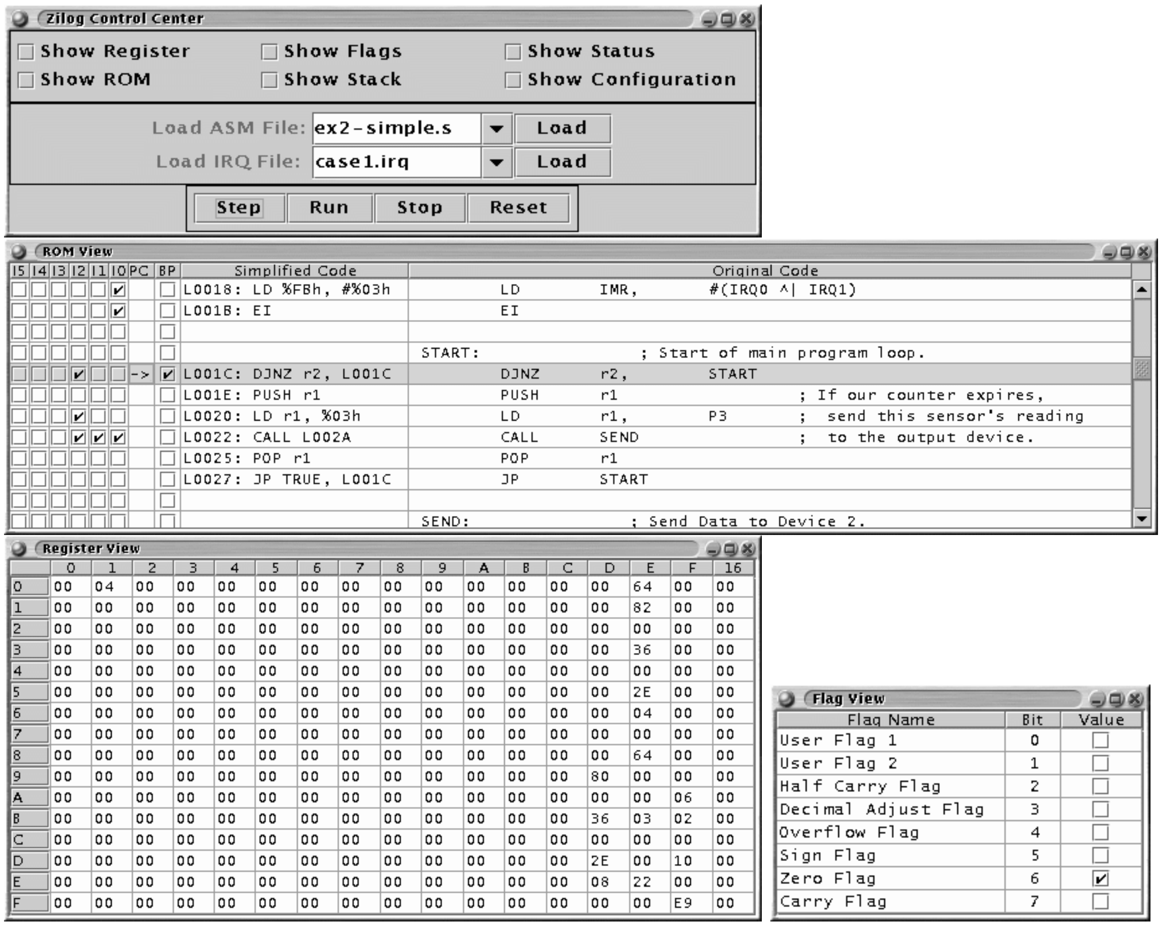
One of the key benefits of the ZARBI simulator is cycle-accurate interrupt behavior. The commercially-available Zilog in-circuit emulator does not allow single-stepping through interrupt handlers and does not maintain correct clock state when single-stepping.

The absence of cycle-accurate simulators and models for many modern processors is a major obstacle to verification of real-time software in practice.

When run in command-line mode, the simulator outputs maximum stack depth for a given run, and may allow device models to output status information to the console.

6.2.3 State Machine Models

All of the commercial benchmarks examined in this dissertation were written for Z86 chips connected to other peripheral devices like analog-to-digital converters, liquid crystal displays, universal asynchronous receiver/transmitters, and external



☐ Show Register ☐ Show Flags ☐ Show Status
☐ Show ROM ☐ Show Stack ☐ Show Configuration

Load ASM File: ex2-simple.s Load
Load IRQ File: case1.irq Load

I5	I4	I3	I2	I1	I0	PC	BP	Simplified Code	Original Code
☐	☐	☐	☐	☐	☑		☐	L0018: LD %FBh, #%03h	LD IMR, #(IRQ0 ^| IRQ1)
☐	☐	☐	☐	☐	☑		☐	L001B: EI	EI
☐	☐	☐	☐	☐	☐		☐		
☐	☐	☐	☐	☐	☐		☐		START: ; Start of main program loop.
☐	☐	☐	☑	☐	☐	->	☑	L001C: DJNZ r2, L001C	DJNZ r2, START
☐	☐	☐	☐	☐	☐		☐	L001E: PUSH r1	PUSH r1 ; If our counter expires,
☐	☐	☐	☑	☐	☐		☐	L0020: LD r1, %03h	LD r1, P3 ; send this sensor's reading
☐	☐	☐	☑	☑	☑		☐	L0022: CALL L002A	CALL SEND ; to the output device.
☐	☐	☐	☐	☐	☐		☐	L0025: POP r1	POP r1
☐	☐	☐	☐	☐	☐		☐	L0027: JP TRUE, L001C	JP START
☐	☐	☐	☐	☐	☐		☐		
☐	☐	☐	☐	☐	☐		☐		SEND: ; Send Data to Device 2.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	16
0	00	04	00	00	00	00	00	00	00	00	00	00	00	00	64	00	00
1	00	00	00	00	00	00	00	00	00	00	00	00	00	00	82	00	00
2	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
3	00	00	00	00	00	00	00	00	00	00	00	00	00	00	36	00	00
4	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
5	00	00	00	00	00	00	00	00	00	00	00	00	00	00	2E	00	00
6	00	00	00	00	00	00	00	00	00	00	00	00	00	00	04	00	00
7	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
8	00	00	00	00	00	00	00	00	00	00	00	00	00	00	64	00	00
9	00	00	00	00	00	00	00	00	00	00	00	00	00	80	00	00	00
A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	06	00
B	00	00	00	00	00	00	00	00	00	00	00	00	00	36	03	02	00
C	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D	00	00	00	00	00	00	00	00	00	00	00	00	00	2E	00	10	00
E	00	00	00	00	00	00	00	00	00	00	00	00	00	08	22	00	00
F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	E9	00

Flag Name	Bit	Value
User Flag 1	0	☐
User Flag 2	1	☐
Half Carry Flag	2	☐
Decimal Adjust Flag	3	☐
Overflow Flag	4	☐
Sign Flag	5	☐
Zero Flag	6	☑
Carry Flag	7	☐

Figure 6.1. Screenshots from the ZARBI Simulator

memory devices. The exact functions of these various devices are unrelated to the Z86 chip or its simulator, but the interactions they provide are necessary for the software to exercise typical control flow paths. For example, most of the benchmarks communicate with an intelligent LCD display over a 4- or 8-bit data bus. If the display does not acknowledge each command, the control software does not proceed to the main operating loop.

In order to address this problem, the ZARBI simulator includes an interface for models of external chips to be plugged into the simulation.

The external device interface allows the simulator to reset external device models, simulating a power cycle, or to pass information about elapsed time in the simulation. In the real system hardware, external devices are not connected to the Z86's internal clock, but the simulator needs to pass a time reference to the device models.

The external device models are implemented as simple state machines in Java code, which was sufficient for all of the external devices found in the benchmark systems. They interact with the simulated Z86 through memory-mapped I/O, just as in the actual systems.

This modular interface allows many kinds of external devices to be simulated and permits the types, versions, and locations of external devices to be reconfigured appropriately for each of the different benchmark systems.

The external device models, in turn, are separated from the concerns of the Z86 model and can perform their own functions. For example, the model of the external LCD device can calculate based on internal state what text would appear on the LCD panel in the real system and pipe this to standard output during simulation.

Accounting for external devices in embedded systems has proven to be an important part of this project, despite the fact that these factors are overlooked in much of the research in embedded systems research.

6.2.4 Genetic Algorithm

Searching for realizable paths that lead to a maximum stack height is intractable in the general case (see Section 3.2). Exhaustive search for such a path is also impractical given the combinatorially explosive size of the state space. Instead, heuristic searches must be relied upon to find realizable paths with large stack heights. Thus, ZARBI employs what is known as a *genetic* or *evolutionary* search algorithm [34] to find tight lower bounds on the maximum stack height of the benchmarks. Genetic algorithms are a complex topic largely beyond the scope of this dissertation, but this section outlines the major parameters supplied to the search heuristic for the sake of completeness.

When performing a genetic algorithm search for interrupt schedules that yield large maximal stack heights, the ZARBI Simulator backend is run without the graphical user interface, in batch mode. A shell script manages the simulator runs, and performs the evolutionary adjustments to the population of interrupt schedules.

Briefly, genetic algorithms use a fitness heuristic to select good solutions out of diverse population of possible solutions. The search then merges and mutates the qualities of good solutions in hopes of finding better solutions.

For the experiments in Chapter 4, the genetic algorithm searches were run for 25 generations, each with a population of 25 interrupt schedules. The initial population was 400 randomly generated interrupt schedules. The simulator was run for 120 seconds of simulated time on each interrupt schedule, and the maximal stack size during the run was taken to be the fitness function for the interrupt schedule. (It is the nature of the commercial benchmarks that they have 1-second long operating cycles. Thus, discounting a few seconds of startup time, it was expected that their behavior would stabilize after each second, so 120 seconds of run time per individual would be quite sufficient to observe maximal stack size under given conditions.)

From each generation to the next, the top three interrupt schedules were automatically passed on to the next generation. The remainder of the new population was generated using two parents selected from the old population using tournament selection. The parental pair of schedules was merged using standard crossover, and subjected to probabilistic mutation.

The full details of interrupt schedules are explained in Appendix C; crossover between two interrupt schedules was defined as three separate random merges between the three distinct classes of interrupt schedule lines.

Overall mutation rate for children was linearly decreasing from 10% in the first generation, down to 4% in the final generation. Each of the three kinds of interrupt schedule lines underwent specific mutation, in order to preserve the sense of its fitness.

The one-shot interrupt lines had a 20% chance of having the IRQ number randomly permuted, and an 80% chance of having the trigger address shifted +/- 3 instructions.

Periodic address-triggered interrupt lines had a 10% chance of IRQ number mutation, a 40% chance of trigger address shifting +/- 3 instructions, and a 40% chance of period mutation by +/- 0.5% of maximum period.

Periodic time-triggered interrupt lines had a 10% chance of IRQ number mutation, a 40% chance of trigger time shifting +/- 0.5%, and a 50% chance of period mutation by +/- 0.5% of maximum period.

The parameters to the genetic search algorithm have not been closely examined in these experiments, but were sufficiently suitable that the genetic search for each benchmark yielded an interrupt schedule with as good or better maximal stack height than manually selected expert interrupt schedules.

6.3 Deadline Analysis Tools

The second half of this chapter concerns implementation details for the major tools used in the deadline analysis phases of ZARBI. Section 6.3.1 presents the implementation of graph coloring, while Section 6.3.2 describes the multiresolution analysis with adaptive slicing. Sections 6.3.3, 6.3.4 and 6.3.5 outline the various graph visualization and debugging mechanisms built into ZARBI.

6.3.1 Coloring Algorithm

The ZARBI algorithm for graph coloring is presented in CTL notation in Figure 5.6. This section details the actual implementation of the coloring decision in pseudocode and explains the coloring traversal.

As the first step, all nodes in the graph are colored red, whether they are reachable or not. This has the natural side effect that nodes which cannot be reached via backward traversal from the interrupt handler will necessarily be red. As a result, there are no rules for deciding to color a node red, because red nodes will not be passed through the decision code.

All nodes corresponding to the first instruction in the interrupt handler of interest are collected into a worklist and colored ultragreen.

A backward coloring traversal continues for as long as the worklist of nodes is empty. Nodes are taken off of the worklist one at a time and are considered for coloring. The coloring decision rules are given in Figure 6.2. After the node is given an initial coloring, all of its incoming edges are visited. If the source node on an incoming edge is not marked, the source node is put on the end of the worklist. After all of the incoming edges have been visited, the current node is marked. (If the node remains marked, this initial coloring will become its final color. Nodes are only unmarked if one of their outgoing edges changes color.) The algorithm then iterates to the next node in the worklist.

When an edge is visited, it is given the initial coloring of its destination node. If this is a recoloring of the edge, the source node of the edge is explicitly unmarked so that it can be re-examined when it is put back on the worklist.

1.	n in first node of greenIRQ handler	$\Rightarrow$	$n \in UltraGreen$
2.	$(\exists e \in \Omega(n)).(e \in UltraGreen)$	$\Rightarrow$	$n \in Green$
3.	$(\forall e \in \Omega(n)).(e \in Green)$	$\Rightarrow$	$n \in Green$
4.	$(\exists e \in \Omega(n)).(Timesum(e) \wedge (e \in Green))$	$\Rightarrow$	$n \in Green$
5.	$(\exists e \in \Omega(n)).(Timesum(e) \wedge (e \in Blue))$	$\Rightarrow$	$n \in Blue$
6.	$((\forall e \in \Omega(n)).(\neg Push3(e) \vee$ $(Push3(e) \wedge (e \notin Red) \wedge (e \notin UltraGreen)))) \wedge$ $(\exists e \in \Omega(n)).(e \in Yellow) \vee (e \in Magenta)$	$\Rightarrow$	$n \in Magenta$
7.	$(\forall e \in \Omega(n).(((e \in Green) \vee (e \in Magenta) \vee$ $(e \in Blue)) \wedge (\neg Push3(e)))$	$\Rightarrow$	$n \in Blue$
8.	Else	$\Rightarrow$	$n \in Yellow$

where $\Omega(n)$ is the set of node n's outgoing edges, and the predicates $Push3(n)$ and $Timesum(n)$ are true for interrupt and time summary edges, respectively.

Figure 6.2. ZARBI Graph Coloring Decision Rules

The coloring algorithm terminates when all node colors stabilize. Termination is guaranteed to take place because of the precedence of the coloring rules. Nodes that are colored green are *done* – will not be recolored to some other color – as it can be shown inductively that all downstream edges and nodes are also done in order for a node to be colored green. Modulo green, magenta nodes are also done – that is, once a node is magenta, it will either stay magenta or eventually become green. Similarly, blue nodes are done modulo green, which leaves only yellow and red. There is no rule for judging a node to be recolored red, so the coloring algorithm will eventually converge and terminate.

6.3.2 Adaptive Slicing

```
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 6.3. Adaptive Slicing Algorithm

The ZARBI 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 5.2.4; the details of the implementation are presented here, with pseudocode shown in Figure 6.3.

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.

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 6.3 – until the list of final slicing candidates is of zero size.

The adaptive slicing algorithm is not optimal in that a lot of work is 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 cleaner algorithm could 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 main 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.

6.3.3 Colordot

One of the daunting practical problems in deadline analysis of real interrupt-driven programs is finding ways to make sense of the enormous amount of data available. When a user is in the process of running the tool to discover yellow loops that require oracle assertions, even a small commercial example can present thousands of lines of code, and potentially hundreds of thousands of nodes and edges to examine. ZARBI provides several output modes which organize the analysis data into different perspectives.

The *colordot* output tool takes advantage of the key observation that like-colored nodes tend to occur in contiguous zones. A colordot dump of the analysis prints out dots for each combination of PC and IMR that exists in the program; the color of the dot is an attempt to summarize the colors of the possibly many nodes in the graph with matching PC/IMR values.

The rationale behind providing this graphical representation is that with relatively little practice, users of ZARBI can quickly learn to identify trouble spots in the graph that will require greater attention.

```
  imr = 00
L000C   ||
L000F   ||
L0012   ||
  imr = 00 81
L0015      ||
L0017      ||
L0019      ||
L001B   ||
L001E   ||
L0020   ||
  imr = 00 81 83
L0023         ||
L0026   ||
L0028   ||
L002A   ||
  imr = 00 81 83 01 03
L002B            || ||
L002C               ||
  imr = 00 81 83 01 03
```

Figure 6.4. Colordot Output for FSE03

The colordot output for FSE03, (described in Figure 5.2,) with the green interrupt defined as IRQ1, is shown in Figure 6.4.

The format of the output can be read as a two-dimensional table, with executable code labels printed in ascending order down the vertical axis, and IMR values printed at regular intervals along the horizontal axis in the order in which they were encountered.

Thus, the program in Figure 6.4 begins with the instruction at label “000C”, with an IMR value of “00”. Scanning linearly down the executable addresses, the first non-zero IMR value of “81” appears at label “0015”, and so on.

A new horizontal legend (starting with “imr = ...”,) is printed each time a new IMR value is encountered, or every 25th line of output, if no changes take place.

A third important axis in the colordot output is not visible in Figure 6.4; as the name implies, the dots in the display are colored. (Also, they are not dots. ASCII pipe characters proved to be more easily visible in long dumps, but the name “colordot”

```
  imr = 00
L000C   BB
L000F   BB
L0012   BB
  imr = 00 81
L0015      MM
L0017      MM
L0019      MM
L001B   YY
L001E   YY
L0020   GG
  imr = 00 81 83
L0023         GG
L0026   BB
L0028   BB
L002A   BB
  imr = 00 81 83 01 03
L002B            BB GG
L002C               XX
  imr = 00 81 83 01 03
```

B = Blue; M = Magenta; Y = Yellow; G = Green; X = Ultra Green

Figure 6.5. Colordot Output for FSE03 with Colors Abbreviated

was already entrenched in the documentation.) Figure 6.5 recasts the output of Figure 6.4 with the bars replaced by color descriptions.

In a color display, the colordot output can be quickly interpreted to see that label 1E is the transition point between well-behaved green nodes and the rest of the graph. Furthermore, it can quickly be seen that the initialization section prior to label 15 probably needs no additional attention.

In the example above, each pair of *dots* is the same color. That is, for any given PC/IMR combination, there are two identical dots. While this is true of all of the toy examples shown in previous chapters. it is not generally true for many nodes in the commercial benchmarks.

In complex graphs where a PC/IMR pair may contain hundreds of nodes differentiated only by stack context, there may be several different colors of nodes present. In these cases, the colordot tool outputs two different colored dots for the *high* and *low* colors of the many nodes at that location. Thus, a PC/IMR pair that has both green and yellow nodes would have a green and a yellow dot in the associated dump. This information is especially useful when scanning the analysis dump looking for subroutines which are called from contexts of several different colors.

The colordot output format is but one possible view of the formidable data available during the ZARBI deadline analysis. It has proven useful in practice for quickly identifying segments of the graph which require additional time summary oracle assertions. However, the colordot output format has several weaknesses which make it necessary to rely on other auxiliary display formats during serious deadline analysis.

Colordot does not show control flow edges, and is therefore difficult to interpret in program areas where dominant control flow is not in a straight line.

Colordot gives no indication of how many nodes are collected together under a given PC/IMR pair, thereby disguising program "hot spots" which often warrant additional manual attention.

The high/low color scheme is not well defined, as there does not appear to be any ordering of the node colors which allows the high/low scheme to provide the best summary information in all desirable contexts. In cases where nodes of more than two colors must be represented, it is not obvious which color should be hidden. Slight perturbations of the colordot high/low preferences can substantially alter the overall appearances of the more complex graphs.

The colordot tool has no provisions for displaying graphs with more IMR values than can conveniently fit across the viewable text terminal. In practice, this has not proven to be a significant limitation.

Despite these shortfalls, colordot has proven to be a quick and effective tool for visualizing the deadline analysis data in the ZARBI prototype.

6.3.4 Graph Crawler

When more detailed visualization of the deadline analysis graph is required than can be provided with the colordot tool, ZARBI provides a *graph crawler*.

The crawler is a command-line interface that allows the user to navigate the nodes and edges of the graph in complete detail. The crawler state machine moves through the graph based upon commands from a command-line interface, as shown in Figure 6.6

A typical crawler interaction on the FSE03 benchmark is shown in Figure 6.7.

The partial transcript in the figure starts at the nexus for label "0023", and displays the corresponding assembly language for reference. The user selects "P" to print the current nexus out, but there is only one node to display. Choosing the only available node, the corresponding graph notation for the node is displayed. The users requests "O" for outgoing edges, and the three outgoing edges from this node are displayed. By selecting one of the edges, the crawler will move on to the destination node of the chosen edge.

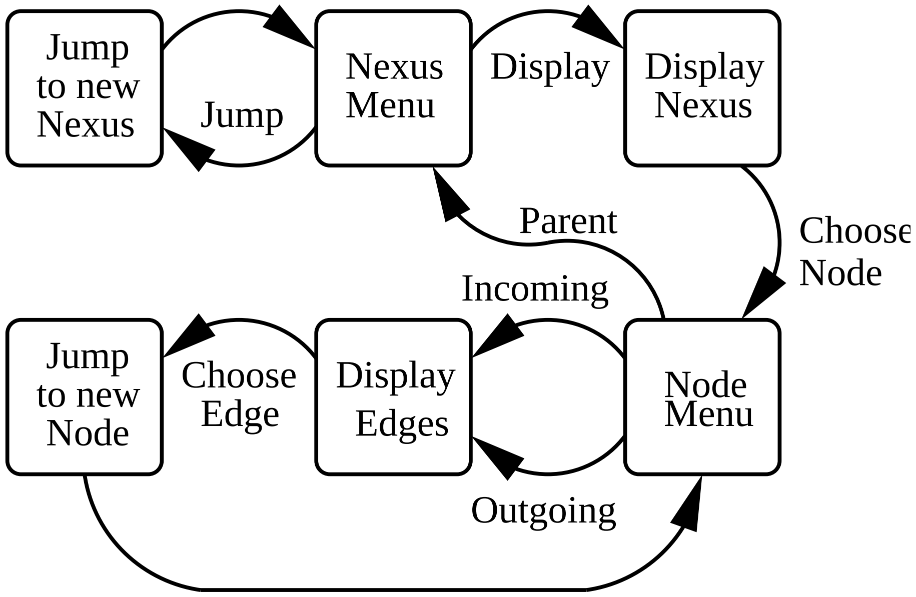


Figure 6.6. Crawler State Machine

The command-line interface displays nodes and edges in full color, corresponding to their final status in the deadline analysis.

The primary disadvantage of the crawler is that it is difficult to visualize the state of nodes that are nearby, but more than one edge away from the current node. Also, the listing of nodes and edges can be many times longer than the available text window in complex graphs; these long listings appear to the untrained eye to be largely identical, making navigation a slow, "crawling" process.

Despite its drawbacks, the crawler makes it possible to drill down into the heart of complex graphs while still being able to visualize the status of adjacent nodes and edges. Experimentally, the crawler has proven essential both during the debugging stages of ZARBI development, and when trying to unravel complex yellow control flow during deadline analysis.

6.3.5 Graph File Format

ZARBI has a flat ASCII file format into which it can dump the final graph, as given by the grammar in Figure 6.8. This format has proven amenable as input to other prototype model checking utilities and visualization tools.

```
Current Nexus is L0023:       JP      TRUE,   L0023
        P       Print current Nexus
        J       Jump to Nexus

        ? p
0       [0x0023,0x83,{}]

        ? 0

        ? Current Node is [0x0023,0x83,{}]
        I       Print current Node incoming edges
        N       Go to parent Nexus
        O       Print current Node outgoing edges

        ? o
0       [0x0023,0x83,{}] -> [0x0023,0x83,{}] = (12,0x00,0x0000)
1       [0x0023,0x83,{}] -> [0x002B,0x03,{0x0023}] = (24,0x13,0x0023)
2       [0x0023,0x83,{}] -> [0x002C,0x03,{0x0023}] = (24,0x13,0x0023)
```

Figure 6.7. Crawler Interface

Goal()	::=	(Edge())* EOF
Edge()	::=	Vertex() → Vertex() = Value()
Vertex()	::=	[Word() , Byte() , { Word() (, Word())* } (, Pair())*]
	||	[Word() , Byte() , { } (, Pair())*]
Value()	::=	“(“ Byte() , Byte() , Word() (, Byte())* “)”
Pair()	::=	Byte() : Byte()
Byte()	::=	An 8-bit quantity.
Word()	::=	A 16-bit quantity.

Figure 6.8. ZARBI Graph File Format

While not used in the current prototype, the format includes provisions for arbitrary pairs of bytes to be appended to each node. This is intended to support slicing and unrolling of loops; current loop nodes would be duplicated and annotated with information of the form, “$rx : y$”, where x would be the loop register, and y would be the precise values it could be unrolled into.

The graph file format output of FSE03, used as the running example throughout Chapters 5 and 6, is shown in Figure 6.9.

6.4 Summary

The Zilog Architecture Resource-Bounding Infrastructure is just that – a collection of tools and data structures that provide general support for control flow graph-based, resource-bounding analyses of Zilog-based microcontroller systems. The current prototype is targeted to the Z86E30, but the backend analysis tools operate on the CFG abstractions presented in Chapter 3 and are less dependent on Z86 assembly syntax.

Components within ZARBI include parsers, a partial compiler, a graph building engine, a simulator, general traversal tools, several kinds of data visualization tools, not to mention the actual stack-size and deadline analysis engines.

Many of the components have been constructed with modularity and future expansion in mind.

```
[0x000C,0x00,{}] -> [0x0026,0x00,{0x000F}] = (20,0x12,0x000F)
[0x000C,0x00,{}] -> [0x000F,0x00,{}] = (00,0x40,0x0000)
[0x0026,0x00,{0x000F}] -> [0x002A,0x00,{0x000F}] = (00,0x40,0x0000)
[0x0026,0x00,{0x000F}] -> [0x0028,0x00,{?,0x000F}] = (10,0x11,0x0000)
[0x000F,0x00,{}] -> [0x0026,0x00,{0x0012}] = (20,0x12,0x0012)
[0x000F,0x00,{}] -> [0x0012,0x00,{}] = (00,0x40,0x0000)
[0x002A,0x00,{0x000F}] -> [0x000F,0x00,{}] = (14,0x22,0x000F)
[0x0028,0x00,{?,0x000F}] -> [0x002A,0x00,{0x000F}] = (10,0x21,0x??)
[0x0026,0x00,{0x0012}] -> [0x002A,0x00,{0x0012}] = (00,0x40,0x0000)
[0x0026,0x00,{0x0012}] -> [0x0028,0x00,{?,0x0012}] = (10,0x11,0x0000)
[0x0012,0x00,{}] -> [0x0015,0x81,{}] = (10,0x00,0x0000)
[0x002A,0x00,{0x0012}] -> [0x0012,0x00,{}] = (14,0x22,0x0012)
[0x0028,0x00,{?,0x0012}] -> [0x002A,0x00,{0x0012}] = (10,0x21,0x??)
[0x0015,0x81,{}] -> [0x0017,0x81,{}] = (10,0x00,0x0000)
[0x0015,0x81,{}] -> [0x002B,0x01,{0x0015}] = (24,0x13,0x0015)
[0x0017,0x81,{}] -> [0x0019,0x81,{}] = (10,0x00,0x0000)
[0x0017,0x81,{}] -> [0x0015,0x81,{}] = (12,0x00,0x0000)
[0x0017,0x81,{}] -> [0x002B,0x01,{0x0017}] = (24,0x13,0x0017)
[0x002B,0x01,{0x0015}] -> [0x0015,0x81,{}] = (16,0x23,0x0015)
[0x0019,0x81,{}] -> [0x001B,0x00,{}] = (06,0x00,0x0000)
[0x0019,0x81,{}] -> [0x002B,0x01,{0x0019}] = (24,0x13,0x0019)
[0x002B,0x01,{0x0017}] -> [0x0017,0x81,{}] = (16,0x23,0x0017)
[0x001B,0x00,{}] -> [0x001E,0x00,{}] = (10,0x00,0x0000)
[0x002B,0x01,{0x0019}] -> [0x0019,0x81,{}] = (16,0x23,0x0019)
[0x001E,0x00,{}] -> [0x0020,0x00,{}] = (10,0x00,0x0000)
[0x001E,0x00,{}] -> [0x001B,0x00,{}] = (12,0x00,0x0000)
[0x0020,0x00,{}] -> [0x0023,0x83,{}] = (10,0x00,0x0000)
[0x0023,0x83,{}] -> [0x0023,0x83,{}] = (12,0x00,0x0000)
[0x0023,0x83,{}] -> [0x002B,0x03,{0x0023}] = (24,0x13,0x0023)
[0x0023,0x83,{}] -> [0x002C,0x03,{0x0023}] = (24,0x13,0x0023)
[0x002B,0x03,{0x0023}] -> [0x0023,0x83,{}] = (16,0x23,0x0023)
[0x002C,0x03,{0x0023}] -> [0x0023,0x83,{}] = (16,0x23,0x0023)
```

Figure 6.9. ZARBI Graph File Format Dump of FSE03

7 SUMMARY AND FUTURE WORK

7.1 Summary

Static checking can provide safe and tight bounds on stack usage and execution times in interrupt-driven systems. This dissertation presents algorithms for resource bound analyses; also presented is ZARBI, a prototype implementation which statically computes stack size and execution time bounds for a benchmark suite of commercially-deployed, interrupt-driven systems. Advanced knowledge of resource bounds enables real-time system designers to eliminate whole classes of errors from their software before testing begins, thereby reducing the testing effort necessary to achieve confidence in their system.

Despite the ubiquity of hardware interrupts in real-time systems, little prior research has dealt with interrupt-driven software. The commercial benchmark suite examined here included proprietary Z86-based microcontrollers programmed in assembly language, with multiple vectored interrupt sources, a shared system stack, extensive use of unstructured loops, and no formal loop annotations.

The stack analysis presented by this dissertation bounds the maximum stack size to within one byte of the true maximum in all but one of the commercial benchmarks. The deadline analysis found firm worst-case latencies in 30% of the cases; in the remaining 70% of the cases, the analysis reduced the size of the testing problem by an average of 98%. While the testing effort still required for these systems is large, it is several orders of magnitude smaller than the testing problem without deadline analysis.

This dissertation presents novel algorithms for bounding stack height and maximum interrupt latency. This is the first such work on tractable control-flow analysis in the presence of vectored interrupt handling.

A secondary contribution of this dissertation is a proof-of-concept implementation of the novel analyses. The implementation is one of the first tools to give an efficient and useful static analysis of assembly code, and the first to analyze interrupt-driven assembly code. The prototype presented here is also among the first to incorporate static analysis with testing oracles in an interactive fashion.

The analysis algorithms also check for several classes of semantic errors in the Z86 program, including using simple types to detect stack manipulation errors. In addition, ZARBI contains components for enhanced visualization and debugging of control-flow graph "problem areas" during the interactive process of interrupt latency analysis.

The current incarnation of the tool uses a Z86 front end, but the abstractions used in the graph analysis are applicable to a wide range of other processors which use bit-maskable, vectored interrupt handling. Examples include the Motorola 68000 family [60, 61], the Intel 8051 family [45], the National Semiconductor COPS8 family [64], as well as several families of special purpose chips.

7.2 Future Work

The success of the analyses presented here paves the way for many areas of potential future work. Other researchers have already begun to reference the paper [14] on ZARBI's stack analysis, and to build this technology into their own analysis tools [63, 82]. Related papers have examined the complexity of the stack analysis first presented here [18], or have rephrased the model checking approach as a type-checking problem [73].

The existence of an analysis infrastructure than can answer questions about WCET in interrupt-driven software enables many new questions to be asked. Many processors used in real-time applications have hardware watch-dog timers – a sort of software dead-man's switch, which resets the processor state if a particular opcode is not executed within a given period. Watch-dog timers are deployed in most of the systems in the ZARBI test suite, but little prior work has addressed the kind of errors that this feature can cause. The techniques presented in this dissertation could be applied to the watchdog timer question; rather than calculating worst-case interrupt latency, the analysis could search for code segments that are more than x cycles away from a `WDT` instruction, where x is the maximum watch-dog period.

Real-time software without interrupts has been analyzed in great depth, and resource bounds can now be calculated for some systems where pipeline and cache effects contribute significantly to WCET analysis. With ZARBI as a baseline, it may be possible to extend these techniques to account for pipeline and cache effects in systems with vectored interrupt handling.

Combining the current analysis with meta-information about minimum interrupt periods and interrupt priority could result in an analysis that would be able to bound resources in more complex systems, or eliminate more magenta nodes in the current systems. Automatic discovery of internal and data-dependent loop bounds would improve the tool's ease of use by inferring many of the assertions that are currently provided to the time summary oracles.

Recursion is uncommon in real-time systems, but not unheard of [25]. Extensions to the analysis algorithms to allow recursive functions would also apply to iterative loops with non-zero stack behavior, as found in "FAN005", the one member of the commercial benchmark suite which has defied analysis thus far.

Any complex analysis of real systems can produce copious amounts of data, as is the case with the deadline analysis presented here. While ZARBI includes several tools to assist the user in visualizing and comprehending this data, better visualization techniques are possible. Work is in progress on a three-dimensional representation of the deadline analysis CFG output, with the goal of making border yellow nodes easier to identify and eliminate.

The dream of building a “push-button” resource-bounding tool for real-time system designers remains beyond the reach of research this author is familiar with. Real-time systems are inherently complex, and the questions a designer would like to ask of an analysis are often highly specific to the given system. Even so, the prototype implementation shown here demonstrates that a general purpose framework can be built which allows someone with expertise in real-time systems to use static analysis without having to become an expert in static analysis. The prototype presented in this dissertation is not industrial strength, but the principles it demonstrates may one day influence real tools. Better tools for bounding resource usage in real-time systems would benefit system designers, and ultimately, consumers of embedded, reactive, and interrupt-driven systems.

LIST OF REFERENCES

LIST OF 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] George S. Avrunin, James C. Corbett, and Laura K. Dillon. Analyzing partially-implemented real-time systems. *IEEE Transactions on Software Engineering*, 24(8):602–614, August 1998.

[8] Iain Bate, Guillem Bernat, and Peter Puschner. Java virtual-machine support for portable worst-case execution-time analysis. In *Proceedings of ISORC 02: Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 83, Washington D.C., USA, April 2002.

[9] William S. Beebee, Jr. and Martin Rinard. An implementation of scoped memory for real-time Java. 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 289–305, Tahoe City, California, October 2001. Springer-Verlag. URL `http://citeseer.nj.nec.com/beebee01implementation.html`.

[10] 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`.

[11] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[12] Greg Bollella, Ben Brosgol, Steve Furr, David Hardin, Peter Dibble, James Gosling, Mark Turnbull, and Rudy Belliardi. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[13] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of OOPSLA 98: 13th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 183–200, 1998. URL `http://www.cis.unisa.edu.au/~pizza/gj/`.

[14] 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`.

[15] Dennis Brylow and Jens Palsberg. Deadline analysis of interrupt-driven software. In *Proceedings of FSE 03: 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Helsinki, Finland, September 2003. URL `http://esecfse.cs.helsinki.fi/`.

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

[17] 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.

[18] Krishnendu Chatterjee, Di Ma, Rupak Majumdar, Tian Zhao, Thomas A. Henzinger, and Jens Palsberg. Stack size analysis of interrupt driven software. In *Proceedings of SAS 03: Tenth Annual International Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 109–126, San Diego, California, June 2003.

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

[20] 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`.

[21] 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`.

[22] Karl Crary and Stephanie Weirich. Resource bound certification. In *Proceedings of POPL 00: 27th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 184–198, 2000. URL `http://citeseer.nj.nec.com/crary00resource.html`.

[23] 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`.

[24] 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.

[25] Jakob Engblom. Static properties of commercial embedded real-time programs, and their implication for worst-case execution time analysis. In *Proceedings of RTAS 99: Fifth IEEE Real-Time Technology and Applications Symposium*, pages 46–55, Vancouver, Canada, June 1999. URL `http://citeseer.nj.nec.com/engblom99static.html`.

[26] Jakob Engblom. On hardware and hardware models for embedded real-time systems. In *Proceedings of RTES 01: IEEE Workshop on Real-Time Embedded Systems*, December 2001. URL `http://citeseer.nj.nec.com/engblom01hardware.html`.

[27] 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`.

[28] Jakob Engblom, Andreas Ermedahl, and Peter Altenbernd. Facilitating worst-case execution time analysis for optimized code. In *Proceedings of ERTS 98: Tenth EuroMicro Workshop on Real-Time Systems*, Berlin, Germany, June 1998. URL `http://citeseer.nj.nec.com/engblom98facilitating.html`.

[29] 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`.

[30] 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`.

[31] 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.

[32] 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.

[33] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[34] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

[35] 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.

[36] 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.

[37] 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`.

[38] 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`.

[39] 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.

[40] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A time-triggered language for embedded programming. 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 166–184, Tahoe City, California, October 2001. Springer-Verlag. URL `http://citeseer.nj.nec.com/henzinger00giotto.html`.

[41] Thomas A. Henzinger and Christoph M. Kirsch. The embedded machine: Predictable, portable real-time code. In *Proceedings of PLDI 02: International Conference on Programming Language Design and Implementation*, pages 315–326. ACM Press, 2002.

[42] Nat Hillary and Ken Madsen. You can't control what you can't measure, or why it's close to impossible to guarantee real-time software performance on a cpu with on-chip cache. In *Proceedings of WCET 02: Second IEEE International Workshop on Worst Case Execution Time Analysis*, pages 45–48, Vienna, Austria, June 2002.

[43] Erik Yu-Shing Hu, Andy Wellings, and Guillem Bernat. A novel gain time reclaiming framework integrating WCET analysis for object-oriented real-time systems. In *Proceedings of WCET 02: Second IEEE International Workshop on Worst Case Execution Time Analysis*, pages 14–20, Vienna, Austria, June 2002.

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

[45] Intel Corporation. *MCS 51 Microcontroller Family User's Manual*. Mt. Prospect, Illinois, February 1994. URL `http://developer.intel.com/design/mcs51/manuals/272383.htm`.

[46] Eugene Kligerman and Alexander D. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):941–949, September 1986.

[47] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of DAC 95: ACM 32nd Design Automation Conference*, pages 456–461, June 1995.

[48] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong-Sang Kim. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, 1995. URL `http://citeseer.nj.nec.com/lim95accurate.html`.

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

[50] Yanhong A. Liu and Gustavo Gómez. Automatic accurate time-bound analysis for high-level languages. In *Proceedings of LCTES 98: ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 31–40. Springer-Verlag, 1998. URL `http://citeseer.nj.nec.com/liu98automatic.html`.

[51] Thomas Lundqvist and Per Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In *Proceedings of LCTES 98: ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1998. URL `http://citeseer.nj.nec.com/lundqvist98integrating.html`.

[52] Sharad Malik, Margaret Martonosi, and Yau-Tsun Steven Li. Static timing analysis of embedded software. In *Proceedings of DAC 97: ACM 34th Design Automation Conference*, pages 147–152, June 1997. URL `http://citeseer.nj.nec.com/malik97static.html`.

[53] 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`.

[54] 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.

[55] 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.

[56] 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`.

[57] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. Technical report, Cornell University, 1999. URL `http://citeseer.nj.nec.com/morrisett99talx.html`.

[58] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Workshop on Types in Compilation*, Kyoto, Japan, March 1998.

[59] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Proceedings of POPL 98: 25th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 85–97, 1998.

[60] Motorola, Inc. *M68000 8-/16-/32 Bit Microprocessor User's Manual*, 9 edition, 1993. URL `http://ebus.motorola.com/brdata/PDFDB/MICROPROCESSORS/32_BIT/68K-COLDFI%RE/M680X0/MC68000UM.pdf`.

[61] Motorola, Inc. *MC68328 DragonBall Microprocessor User's Manual (preliminary)*, November 1997. URL `http://ebus.motorola.com/brdata/PDFDB/MICROPROCESSORS/32_BIT/68K-COLDFI%RE/M683XX/MC68328P.pdf`.

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

[63] Mayur Naik and Jens Palsberg. Compiling with code-size constraints. In *Proceedings of LCTES 02: ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems, joint with SCOPES 02: Software and Compilers for Embedded Systems*, pages 120–129. ACM Press, June 2002. URL `http://doi.acm.org/10.1145/513829.513851`.

[64] National Semiconductor Corporation. *COP8SBR9/COP8SCR9/COP8SDR9 8-Bit CMOS Flash Based Microcontroller with 32k Memory, Virtual EEPROM and Brownout*. Santa Clara, California, April 2002. URL `http://www.national.com/ds.cgi/CO/COP8SBR9.pdf`.

[65] 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.

[66] 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`.

[67] George Necula. Proof-carrying code. In *Proceedings of POPL 97: 24th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[68] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of PLDI 98: ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, 1998.

[69] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of OSDI 96: Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Berkeley, California, 1996. USENIX. URL `http://citeseer.nj.nec.com/article/necula96safe.html`.

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

[71] Jurg Nievergelt and Klaus H. Hinrichs. *Algorithms & Data Structures With Applications To Graphics and Geometry*. Prentice-Hall, 1993.

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

[73] Jens Palsberg and Di Ma. A typed interrupt calculus. In *Proceedings of FTRTFT 02: Seventh International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 291–310, Oldenburg, Germany, September 2002. Springer-Verlag. URL `http://www.springer.de/cgi-bin/search_book.pl?isbn=3-540-44165-4`.

[74] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(4):576–599, July 1995. URL `http://doi.acm.org/10.1145/210184.210187`.

[75] 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`.

[76] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995. URL `http://citeseer.nj.nec.com/palsberg95safety.html`.

[77] 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`.

[78] 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.

[79] 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`.

[80] 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/`.

[81] 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`.

[82] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. In *Proceedings of EMSOFT 03: Third International Conference on Embedded Software*, 2003. to appear.

[83] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701–726, November 1998. URL `http://www.cs.wisc.edu/wpis/papers/tr1386.ps`.

[84] 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`.

[85] 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`.

[86] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

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

[88] 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.

[89] Alan C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989. URL `http://citeseer.nj.nec.com/shaw89reasoning.html`.

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

[91] Kevin Tao, Wanjun Wang, and Jens Palsberg. Java tree builder. Code available for download, 1997. URL `http://www.cs.purdue.edu/jtb/`.

[92] David Tennenhouse. Intel developer forum. Keynote address transcript, August 2001. San Jose, California.

[93] 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`.

[94] 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`.

[95] 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`.

[96] Sreeni Viswanadha, Sriram Sankar, and Sun Microsystems. Java compiler compiler. Code available for download, 1997. URL `http://www.webgain.com/products/java_cc/`.

[97] 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.

[98] 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.

[99] 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`.

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

APPENDICES

APPENDIX A MICRO00 EXAMPLE PROGRAM

Much of the benchmark suite used throughout this dissertation is either proprietary code which cannot be published, or toy examples which have been presented without sufficient detail to be actual Z86 programs.

For completeness, this appendix presents the "Micro00" benchmark in its entirety. While still a small toy problem, the code is complete – it can be compiled to Z86 object code, burned to a Z86E30 "one-time programmable" chip, and run on bare hardware.

A.1 Example System Overview

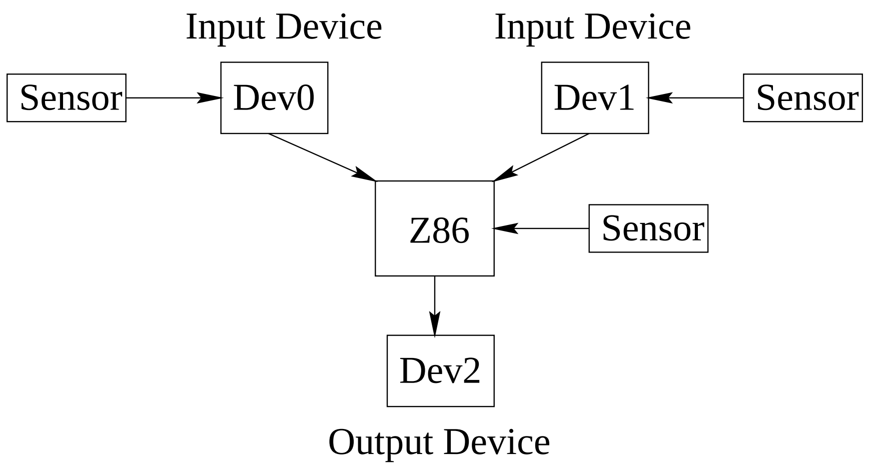


Figure A.1. Conceptual Diagram for Micro00 Example

The example system has three external devices and three sensors as illustrated in Figure A.1. The example hardware is wired as shown in Figure A.2. For clarity of presentation, the text will not dwell on the details of the other three devices in the system. All that matters is that Device 0 and Device 1 have some kind of data that they regularly pass to the Controller. The Controller forwards this data, along with some of its own, to Device 2, which could be any kind of output device.

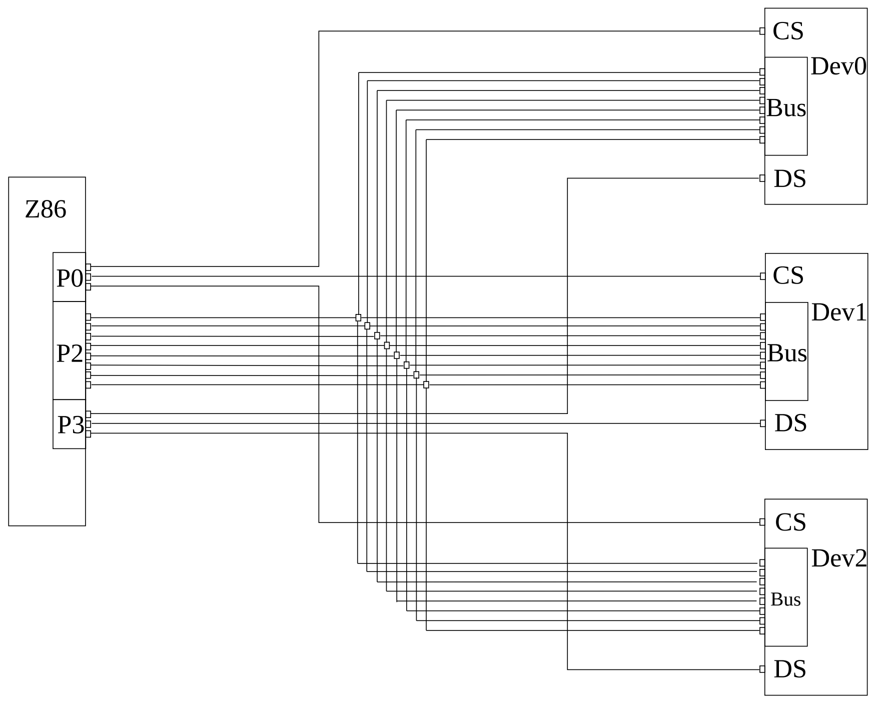


Figure A.2. Hardware Configuration for Micro00 Example System

The electrical protocol observed by these devices is simple. When a device wishes to relay data to the controller, it requests an interrupt with its Data Strobe line. The controller indicates that it is ready by strobing the corresponding device's Chip Select line. The Controller is the "Bus Master", and devices do not speak unless spoken to, via the CS line.

If actually deployed, this type of configuration could be seen in a hierarchical arrangement of environmental controls, where each separate controller relays its sensor data to a logging entity.

Figure A.3 explains the 12-instruction subset of the Z86 assembly language used for the Micro00 example software.

Twenty of the Z86's 256 registers have special purposes, such as port I/O, timer control, or stack management. The relevant special register identifiers for this example program are listed in Figure A.4.

When an interrupt arrives, the controller clears bit 7 of the IMR register, (the equivalent of a DI instruction), and jumps to the correct handler. An IRET instruction sets the bit again, (the equivalent of an EI instruction.)

AND	src, dst	Binary AND the *src* and *dst*, store result in *dst*.
CALL	label	Call procedure. Stores return address on stack, and jumps to *label*.
DI		Disable Interrupts.
DJNZ	dst, label	Decrement, Jump Not Zero. Decrements *dst* register, and jumps to *label* if result is non-zero.
EI		Enable Interrupts.
IRET		Return from Interrupt Handler. Pops condition codes and return address off of stack, continues execution.
JR	cc, label	Jump Relative. If condition code *cc* is true, jumps to *label*. If omitted, *cc* is assumed *true*.
LD	dst, src	Loads register *src* into *dst*.
POP	dst	Pops value off of stack, into dst. (dst = reg[SP++])
PUSH	src	Pushes src register onto stack. (reg[–SP] = src)
RET		Return from procedure. Pops return address off of stack, and continues.
TM	dst, src	Test Mask. Binary AND's the *src* and *dst*; affects condition codes, but does not store result.

Figure A.3. Z86 Instructions Used in Micro00 Example System

Note that there are two return instructions, "RET" and "IRET". "RET" corresponds to the "CALL" instruction; "CALL" and "RET" procedure calls do not guarantee preservation of any registers across the call. The "IRET" instruction, however, does not correspond to an explicit "CALL" instruction. The Z86 has true vectored interrupt handling, which means that control can transfer to any interrupt handler after any opcode, given that the interrupts are enabled. Interrupt handlers preserve the processor condition code register on the stack, but otherwise do not guarantee any other register to be preserved across the call. This means that programmers must work carefully to ensure that their interrupt handlers do not corrupt state information during delicate computations in the non-interrupt code.

A.2 Example System Program

The overall structure of the example program is illustrated by the partial call graph in Figure A.5. The figure does not show control-flow transfers due to interrupts.

Figures A.6, A.7 and A.8 show the Micro00 example program. The program makes communication between the controller and the three devices possible. After a brief initialization segment from lines 17 - 23, the program enters an infinite loop, from which it occasionally breaks in order to relay sensor data from Port 3 (line 27) to Device 2.

IMR	Interrupt Mask Register	Bits 0-5 individually enable each of the 6 interrupt sources. Bit 7 enables vectored interrupt processing. Bit 7 normally enabled with the EI instruction, and disabled with DI.
IRQ	Interrupt Request Register	When interrupt signals arrive, the corresponding bits in the IRQ register are set. This allows interrupts to be handled via polling, and makes visible pending, disabled interrupts.
P0	Port 0	In the example, Port 0 connects to Chip Select lines on each of the three external devices.
P2	Port 2	In the example, Port 2 is the 8-bit data bus connecting the controller to all of the external devices.
P2M	Port 2 Mode Control	P2M allows each of the lines on P2 to be configured as input (data in to Z86), or output (data out of Z86.)
P3	Port 3	In the example, Port 3 is connected to the Data Ready strobes for the external devices. This means that Device 0 can raise Interrupt Request 0, and Device 1 can raise IRQ1.
RP	Register Pointer	Selects register bank for local addressing mode.
SPL	Stack Pointer	Stores Stack Pointer value to be used for internal stack.

Figure A.4. Z86 Special Registers Used in Micro00 Example System

When the main loop calls SEND (line 28), SEND in turn calls DEVOUT (line 35), which calls PULSE (line 40). In the PULSE procedure, interrupts are globally disabled with the DI instruction (line 55) prior to initiating the Chip Select pulse to the output device. This operation must not be interrupted, because this could result in confusing signals being sent to the output device. After the pulse is complete, interrupts are re-enabled by DEVOUT (line 42), and the main loop continues on its merry way.

In the background, vectored interrupt handlers IRQ0 and IRQ1 wait for data to come in from either of the other two devices. When it does, the appropriate handler saves all of the important state registers on the stack, and calls the SEND procedure to relay the data to Device 2.

Note that IRQ1 defers to IRQ0 in lines 76 and 77, dropping the data from Device 1 if Device 0 already has data waiting.

A cursory analysis of the control flow of the code shows that when the EI at line 42 is reached, any one of the four possible combinations of IRQ0 and IRQ1 could be enabled. Even in this small example, it is not immediately clear whether or not the correct combination is always present. The SEND procedure can be called from the main loop, or from IRQVC0 or from IRQVC1; further, under certain circumstances, it can be called from IRQVC0 from within IRQVC1, or from IRQVC1, from within

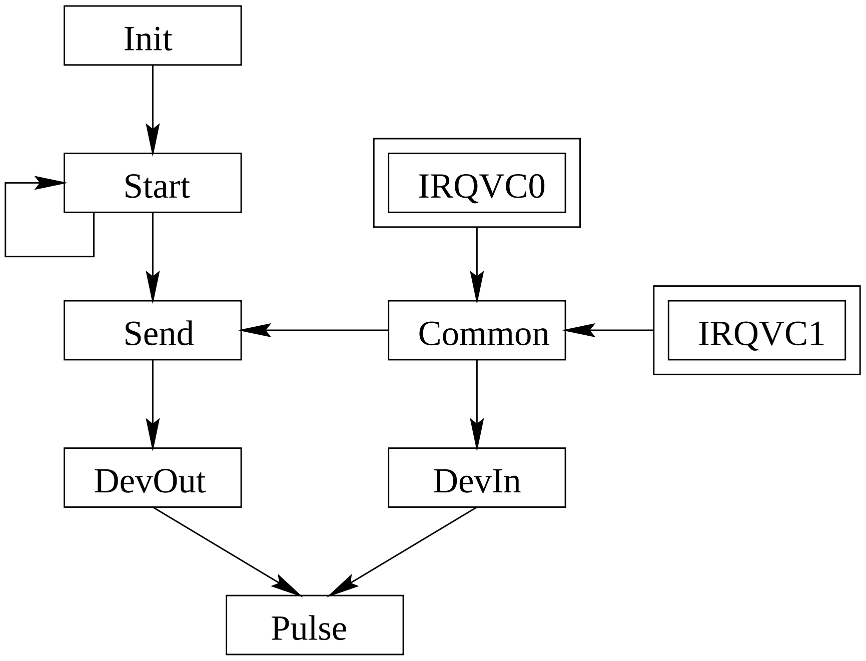


Figure A.5. Partial Call graph for the Micro00 Example Program

IRQVC0. To make things worse, the double interrupt case can take place when the main program is already several CALL levels down into the SEND sequence.

As though static analysis of this example code were not difficult enough to begin with, testing the maximal stack size by simulating interrupts is not straight forward either. Interrupt handlers can have subtle additive and subtractive interactions. As shown by IRQ1 deferring to IRQ0 at line 77, simulating all interrupts firing as often as possible does not necessarily yield the maximum stack size. In practice, interrupts often represent error conditions of some kind, and their handlers can act to slow down normal computation or adjust the stack size arbitrarily.

In short, reliable, precise analysis of the maximal possible stack size, even in relatively small programs, makes for a challenging problem.

A.3 ZARBI results

The ZARBI stack height analysis returns the results shown in Figure A.9 for the Micro00 benchmark.

```
; Constant Pool (Symbol Table).
        ; Bit Flags for IMR and IRQ registers.
IRQ0    .EQU    #00000001b
IRQ1    .EQU    #00000010b
        ; Bit Flags for external devices on Port 0 and Port 3.
DEV0    .EQU    #00000100b
DEV1    .EQU    #00001000b
DEV2    .EQU    #00010000b

; Interrupt Vectors.
        .ORG  %00h
        .WORD #IRQVC0           ; Device 0
        .WORD #IRQVC1           ; Device 1

; Main Program Code.
        .ORG  0Ch
INIT:           ; Initialization section.
        LD      SPL,    #0F0h   ; Initialize Stack Pointer.
        LD      RP,     #10h    ; Work in register bank 1.
        LD      P2M,    #00h    ; Set Port 2 to all outputs.
        LD      IRQ,    #00h    ; Clear any interrupt requests.
        LD      IMR,    #(IRQ0 ^| IRQ1)
        EI                      ; Enable Interrupts 0 and 1.
START:          ; Start of main program loop.
        DJNZ    r2,     START
        PUSH    r1              ; If our counter expires,
        LD      r1,     P3      ;  send this sensor's reading
        CALL    SEND            ;  to the output device.
        POP     r1
        JR      START

```

Figure A.6. Micro00 Example Program

```
SEND:           ; Send Data to Device 2.
        PUSH    r0              ; Save r0 on Stack.
        LD      r0,     #DEV2   ; Select control line for Dev 2.
        CALL    DEVOUT          ; Send out to Device.
        POP     r0              ; Restore r0 to value before
        RET                     ;  SEND was called.
DEVOUT:         ; Send data out to a Device.
        LD      P2,     r1      ; Output data.
        CALL    PULSE           ; Pulse device control line to
                                ;  inform device data awaits.
        EI                      ; Reactivate interrupts,
        RET                     ;  if disabled.
DEVIN:          ; Receive data from a Device.
        DI                      ; Disable interrupts.
        LD      P2M,    #0FFh   ; Set Port 2 lines to all inputs.
        CALL    PULSE           ; Pulse control line to inform
                                ;  device controller awaits data.
        LD      r1,     P2      ; Input data.
        LD      P2M,    #00h    ; Set Port 2 lines to all outputs.
        EI                      ; Reactivate interrupts.
        RET
PULSE:          ; Pulse control line of a device.
        PUSH    IMR             ; Remember interrupt mask.
        DI                      ; Musn't interrupt during pulse.
        LD      P0,     r0      ; Control line determined by r0.
        LD      P0,     #00h
        POP     IMR             ; Reactivate interrupts.
        RET

```

Figure A.7. Micro00 Example Program (continued)

```
IRQVC0:         ; Interrupt for Device 0.
        PUSH    IMR
        AND     IMR,#^C IRQ0    ; Ensure interrupt is not re-fired.
        PUSH    r0              ; Save registers from squashing.
        LD      r0,     #DEV0
COMMON: PUSH    r1
        LD      r2,     #00h    ; Reset counter in main loop.
        CALL    DEVIN
        CALL    SEND
        POP     r1              ; Restore all the saved registers,
        POP     r0              ;  including the IMR,
        POP     IMR             ;  to their pre-interrupt values.
IRQDN:  IRET                    ; Interrupt Handler is done.

IRQVC1:         ; Interrupt for Device 1.
        TM      IRQ,    #IRQ0   ; If Interrupt 0 already pending,
        JR      NZ,     IRQDN   ;  Cancel this handler.
        PUSH    IMR
        AND     IMR,#^C IRQ1    ; Ensure interrupt is not re-fired.
        PUSH    r0              ; Save registers from squashing.
        LD      r0,     #DEV1
        JR      COMMON
.END
```

Figure A.8. Micro00 Example Program (continued)

```
Max Stack Height = 37 at [0x004A,0x80,{0x80}]
                (Guessing path)
                [0x004A,0x80,{0x80}]
                [0x0048,0x80,{0x0038}]
                [0x0035,0x80,{0x0030}]
                [0x002D,0x80,{?}]
                [0x0029,0x80,{0x0066}]
                [0x0063,0x80,{?}]
                [0x005C,0x00,{?}]
                [0x0077,0x00,{0x02}]
                [0x0072,0x02,{0x0047}]
                [0x0047,0x82,{0x0063}]
                [0x0060,0x02,{?}]
                [0x005C,0x02,{?}]
                [0x0058,0x02,{0x03}]
                [0x0053,0x03,{0x001C}]
                [0x001C,0x83,{}]
Coloring graph, IRQ=0:                                      [   OK   ]
Edges = 619     Green   Yellow  Magenta Blue    Red
Nodes = 339     191     0       17      131     0
Percent =       56%     0%      5%      38%     0%
```

Figure A.9. Micro00 Example Program Stack Height Results

APPENDIX B SIMPLIFIED Z86 GRAMMAR

The ZARBI Simplifier takes as input the Z86 Assembly Language described in [100] and emits syntax compliant with the grammar below. The many other ZARBI tools parse in this stricter grammar, thereby avoiding duplicated work like symbol table resolution.

```
       Goal()  ::=  Line() Goal()
       Goal()  ::=  Code() EOF
       Code()  ::=  LabelDef() Line() Code()
                ||  .END
       Line()  ::=  Directive()
                ||  Instruction()
   LabelDef()  ::=  Label() :
Instruction()  ::=  CLR()
                ||  LD()
                ||  LDL()
                ||  LDC()
                ||  POP()
                ||  PUSH()
                ||  ADC()
                ||  ADD()
                ||  CP()
                ||  DA()
                ||  DEC()
                ||  DECW()
||  INC()
||  INCW()
||  SBC()
||  SUB()
||  AND()
||  COM()
||  OR()
||  XOR()
||  CALL()
||  DJNZ()
||  IRET()
||  JP()
||  JR()
||  RET()
||  TCM()
||  TM()
||  LDCI()
||  RL()
||  RLC()
||  RR()
||  RRC()
||  SRA()
||  SWAP()
||  CCF()
||  DI()
||  EI()
||  HALT()
||  NOP()
||  RCF()
||  SCF()
||  SRP()
```

```
                ||    STOP()
                ||    WDH()
                ||    WDT()
     IRET()    ::=    IRET
      RET()    ::=    RET
      CCF()    ::=    CCF
       DI()    ::=    DI
       EI()    ::=    EI
     HALT()    ::=    HALT
      NOP()    ::=    NOP
      RCF()    ::=    RCF
      SCF()    ::=    SCF
     STOP()    ::=    STOP
      WDH()    ::=    WDH
      WDT()    ::=    WDT
      INC()    ::=    INC G_IA_Operand()
     INCW()    ::=    INCW rr Dec_Reg_Pair()
     CALL()    ::=    CALL Label()
                ||    CALL @ rr Dec_Reg_Pair()
                ||    CALL @ CharSymbol()
      CLR()    ::=    CLR G_IA_Operand()
      COM()    ::=    COM G_IA_Operand()
       DA()    ::=    DA G_IA_Operand()
      DEC()    ::=    DEC G_IA_Operand()
     DECW()    ::=    DECW rr Dec_Reg_Pair()
      POP()    ::=    POP G_IA_Operand()
     PUSH()    ::=    PUSH G_IA_Operand()
       RL()    ::=    RL G_IA_Operand()
      RLC()    ::=    RLC G_IA_Operand()
```

```
    RR()  ::=  RR G_IA_Operand()
   RRC()  ::=  RRC G_IA_Operand()
   SRA()  ::=  SRA G_IA_Operand()
  SWAP()  ::=  SWAP G_IA_Operand()
   SRP()  ::=  SRP # CharSymbol()
    JP()  ::=  JP Condition() , Label()
           ||  JP Condition() , rr Dec_Reg_Pair()
           ||  JP Condition() , CharSymbol()
    JR()  ::=  JR Condition() , Label()
  DJNZ()  ::=  DJNZ r Dec_Reg() , Label()
   AND()  ::=  AND AND_Operand()
    OR()  ::=  OR AND_Operand()
   XOR()  ::=  XOR AND_Operand()
   ADD()  ::=  ADD AND_Operand()
   SUB()  ::=  SUB AND_Operand()
   ADC()  ::=  ADC AND_Operand()
   SBC()  ::=  SBC AND_Operand()
    CP()  ::=  CP AND_Operand()
   TCM()  ::=  TCM AND_Operand()
    TM()  ::=  TM AND_Operand()
    LD()  ::=  LD AND_Operand()
           ||  LD @ r Dec_Reg() , r Dec_Reg()
           ||  LD @ CharSymbol() , r Dec_Reg()
           ||  LD @ r Dec_Reg() , CharSymbol()
           ||  LD @ CharSymbol() , CharSymbol()
   LDL()  ::=  LDL rr Dec_Reg_Pair() , Label()
           ||  LDL CharSymbol() , Label()
   LDC()  ::=  LDC r Dec_Reg() , @ rr Dec_Reg_Pair()
           ||  LDC r Dec_Reg() , @ CharSymbol()
```

```
         LDCI()  ::=  LDCI @ r Dec_Reg() , @ rr Dec_Reg_Pair()
                  ||  LDCI @ r Dec_Reg() , @ CharSymbol()
                  ||  LDCI @ CharSymbol() , @ rr Dec_Reg_Pair()
                  ||  LDCI @ CharSymbol() , @ CharSymbol()
 G_IA_Operand()  ::=  r Dec_Reg()
                  ||  CharSymbol()
                  ||  @ r Dec_Reg()
                  ||  @ CharSymbol()
  AND_Operand()  ::=  @ r Dec_Reg() , # CharSymbol()
                  ||  @ CharSymbol() , # CharSymbol()
                  ||  r Dec_Reg() , @ r Dec_Reg()
                  ||  r Dec_Reg() , @ CharSymbol()
                  ||  CharSymbol() , @ r Dec_Reg()
                  ||  CharSymbol() , @ CharSymbol()
                  ||  r Dec_Reg() , # CharSymbol()
                  ||  CharSymbol() , # CharSymbol()
                  ||  r Dec_Reg() , r Dec_Reg()
                  ||  r Dec_Reg() , CharSymbol()
                  ||  CharSymbol() , r Dec_Reg()
                  ||  CharSymbol() , CharSymbol()
    Condition()  ::=  F
                  ||  C
                  ||  NC
                  ||  Z
                  ||  NZ
                  ||  PL
                  ||  MI
                  ||  OV
                  ||  NOV
```

```
                      ||    EQ
                      ||    NE
                      ||    GE
                      ||    GT
                      ||    LE
                      ||    LT
                      ||    UGE
                      ||    ULE
                      ||    ULT
                      ||    UGT
                      ||    TRUE
       Directive()   ::=    . WORD # Label()
                      ||    . ASCII # CharSymbol()
      CharSymbol()   ::=    % Hex_h()
                      ||    % Bin_b()
                      ||    % Dec_d()
           Label()   ::=    An identifier.
         Dec_Reg()   ::=    A register in decimal notation.
    Dec_Reg_Pair()   ::=    A register pair in decimal notation.
           Bin_b()   ::=    An integer in binary notation.
           Dec_d()   ::=    An integer in decimal notation.
           Hex_h()   ::=    An integer in hexadecimal notation.
```

APPENDIX C INTERRUPT SCHEDULE FILE FORMAT

The ZARBI Simulator accepts as input three kinds of interrupt sequences that can be specified in an interrupt schedule file. This appendix gives an example of the interrupt schedule files used for the genetic algorithm search used to find lower bounds on maximum stack heights.

The first kind of interrupt sequence that can be specified indicates that a particular interrupt will fire just before a certain address in the program is executed. The interrupt will fire each time this address is about to be executed.

The other two kinds if interrupt sequences are periodic. They will start firing a specified time after the start of the program or just before a certain address is executed. Besides an initial firing point, these interrupt sequences specify a period in clock cycles.

An interrupt schedule file contains an arbitrary number of single-shot interrupt sequences and periodic interrupt sequences, as illustrated by Figure C.1.

Single-shot interrupt sequences are in the first block of Figure C.1. `IRQ` means interrupt and `ADDR` means address. All addresses are in hexadecimal. The first line says interrupt number 5 will be fired each time the simulator is about to execute the instruction at address `00E4`.

The second block in Figure C.1 contains the periodic interrupt sequences that are started just before a specified address is executed. The number after `EACH` specifies the number of cycles before the interrupt should be fired again. The first of these interrupt sequences specifies that interrupt 5 will fire just before address `04BC` is executed the first time, and then subsequently every 35,721 clock cycles after that.

The third block in Figure C.1 contains the periodic interrupt sequences that are started a fixed number of cycles after the controller begins program execution. The first of these interrupt sequences says that interrupt 1 will fire after the first 703,529 cycles of execution, and will subsequently fire again every 700,748 cycles.

```
IRQ 5 @ ADDR 00E4
IRQ 5 @ ADDR 05D6
IRQ 1 @ ADDR 0298
IRQ 4 @ ADDR 03D5
IRQ 4 @ ADDR 0710

IRQ 5 @ ADDR 04BC EACH 35721
IRQ 4 @ ADDR 00D5 EACH 511911
IRQ 1 @ ADDR 0620 EACH 617499
IRQ 0 @ ADDR 0B2A EACH 254317
IRQ 4 @ ADDR 0A1D EACH 366248

IRQ 1 @ TIME 703529 EACH 700748
IRQ 1 @ TIME 418949 EACH 754244
IRQ 2 @ TIME 701474 EACH 978065
IRQ 4 @ TIME 424882 EACH 601242
IRQ 2 @ TIME 193234 EACH 317528
```

Figure C.1. Example Interrupt Schedule

APPENDIX D FLOW ORACLE GRAMMAR

The ZARBI graph builder uses a "flow oracle" to answer questions about the very small number of indirect jumps contained in the commercial benchmarks. The grammar accepted by the ZARBI Flow Oracle is shown in Figure D.1.

Goal()	::=	Line() Goal()
	||	EOF
Line()	::=	Reg() @ Label() : Info()
Info()	::=	Item() , Info()
	||	Item()
Item()	::=	Range()
	||	Loop()
	||	Atom()
Atom()	::=	Label()
	||	Hex()
Loop()	::=	Hex() TO Hex()
	||	Hex() DOWNTO Hex()
Range()	::=	Hex() .. Hex()
Label()	::=	The syntax of a Z86 program label
Hex()	::=	8-bit, unsigned integers in hexadecimal
Reg()	::=	Z86 assembly syntax for a register

Figure D.1. Flow Oracle Input Grammar

In practice, the current version of the flow oracle is used only to provide lists of possible target addresses for indirect jump instructions. Only one of the seven commercial benchmarks, "Fan005", used indirect jumps at all. In Fan005, the register with the jump target was loaded only with immediate constants, and no pointer arithmetic was performed on its values. Calculating all possible indirect jump targets for a Z86 program is an infeasible data flow analysis problem in the general case, but becomes feasible when the expressive power of the language is sufficiently constrained.

Because data flow analysis is tangential to the primary thrust of this dissertation, the current version of ZARBI takes a shortcut around the problem, and allows "manual" data flow analysis to be specified through the flow oracle. Figure D.2 shows the only flow information used in the experiments presented in this dissertation.

```
%AEh @ L0AC8:   {L0ACB, L0ADD, L0AF6, L0B08,  L0B67, L0B7E, L0B88}
```

Figure D.2. Flow Oracle Input Example

The flow information provided by Figure D.2 states that register pair "%AEh" will contain one of the seven address labels to the right of the colon at program point "L0AC8". This flow information is provided to the graph builder for the Fan005 benchmark, which allows the proper edges to be constructed when the analysis reaches the indirect call instruction at address 0x0AC8 in the program.

Information provided by the flow oracle must be safe in order for the entire deadline and stack-size analyses to be safe. Automating conservative data flow analysis of this kind is beyond the scope of this dissertation.

The flow oracle's syntax was designed to allow flow information for loop variables to be passed to the graph builder for automated loop unrolling, but this has not been implemented in the current version of ZARBI. Automated unrolling of internally-bounded loops could eliminate up to two thirds of the time oracle assertions provided for deadline analysis, as section 5.2.3 described.

The syntax provided by the flow oracle would allow flow information of the form shown in Figure D.3, which states that the loop variable in register 12 at instruction 345 goes from 5 down to 1.

```
%12h @ L0345:   05 DOWNTO 01
```

Figure D.3. Flow Oracle Loop Bound Syntax

The flow oracle interface has allowed manual data flow analysis to be used in the prototype system, but would permit automated analyses to interact with the graph builder in the same fashion.

APPENDIX E TIME ORACLE GRAMMAR

The ZARBI graph builder uses a “time oracle” to answer questions about maximum latency while building the deadline analysis graph. The grammar accepted by the ZARBI Time Oracle is shown in Figure E.1.

Concrete examples of input to the time oracle are provided throughout section 5.4.

```
      Goal()  ::=  Line() Goal()
               ||  EOF
      Line()  ::=  GraphNode() → GraphNode() = Int()
 GraphNode()  ::=  [ Label(), Mask(), StackList() ]
      Mask()  ::=  Var()
               ||  Hex()
 StackList()  ::=  Var()
               ||  { Label() }
               ||  { Hex() }
     Label()  ::=  16-bit, unsigned integers in hexadecimal
       Hex()  ::=  8-bit, unsigned integers in hexadecimal
       Var()  ::=  alphabetic variable name
```

Figure E.1. Time Oracle Input Grammar

VITA

VITA

Dennis William Brylow was born in Milwaukee, and grew up in Greendale, Wisconsin.

Under the tutelage of Gordon Kraemer and his successors at the Greendale High School educational access TV station, Dennis learned about television production, audio and video technology, electronics and computers. He designed and fabricated his first circuit boards the summer of 1989.

Dennis graduated from Rose-Hulman Institute of Technology in 1996 with Bachelor of Science degrees in computer science and electrical engineering. As an undergraduate, his side jobs included lab assistant, documentation manager, and system administrator. In his spare time, he worked for the institute's solar race car project, where he specialized in embedded radio telemetry systems. He spent the summer of 1996 studying abroad in Kanazawa, Japan.

Returning to America, Dennis worked full time for Greenhill Manufacturing, Ltd. Having spent four previous summers at the small company, he quickly became an R & D engineer, participating in all facets of planning, designing, prototyping, programming, fabricating and testing of embedded control systems.

Dennis began graduate study at Purdue University in 1997, where he spent semesters as a teaching assistant, course coordinator, and eventually primary instructor for introductory programming courses. In his spare time, he integrated Linux workstations into the department computing infrastructure and designed circuit boards for specialty instructional laboratories.

As a research assistant under Jens Palsberg, Dennis earned his Master of Science in 1999, and his Doctor of Philosophy in computer science in August of 2003.

His interests include real-time, embedded, and interrupt-driven systems, software engineering, type systems, and UNIX system administration.