Static Checking of Interrupt-driven Software

Dennis Brylow Niels Damgaard Jens Palsberg

Purdue University
Department of Computer Science
West Lafayette, Indiana 47907, USA
{brylow,damgaard,palsberg}@cs.purdue.edu

Abstract

Resource-constrained devices are becoming ubiquitous. Examples include cell phones, palm pilots, and digital thermostats. It can be difficult to fit required functionality into such a device without sacrificing the simplicity and clarity of the software. Increasingly complex embedded systems require extensive brute-force testing, making development and maintenance costly. This is particularly true for system components that are written in assembly language. Static checking has the potential of alleviating these problems, but until now there has been little tool support for programming at the assembly level.

In this paper we present the design and implementation of a static checker for interrupt-driven Z86-based software with hard real-time requirements. For six commercial microcontrollers, our checker has produced upper bounds on interrupt latencies and stack sizes, as well as verified fundamental safety and liveness properties. Our approach is based on a known algorithm for model checking of pushdown systems, and produces a control-flow graph annotated with information about time, space, safety, and liveness. Each benchmark is approximately 1000 lines of code, and the checking is done in a few seconds on a standard PC. Our tool is one of the first to give an efficient and useful static analysis of assembly code. It enables increased confidence in correctness, significantly reduced testing requirements, and support for maintenance throughout the system life-cycle.

1 INTRODUCTION

Overview

Resource-constrained devices are becoming ubiquitous. Examples include cell phones, Palm Pilots, and digital thermostats. It can be difficult to fit required functionality into such a device without sacrificing the simplicity and clarity of the software. Increasingly complex embedded systems require extensive brute-force testing, making development and maintenance costly.

Our focus is on interrupt-oriented small devices. For example, consider the Z86E30 processor [18], 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 ports. In addition, the Z86 has six levels of vectored interrupt processing, and two internal timers. Despite the Z86's meager resources, it is deployed in a daunting array of elaborate systems, where larger, more powerful processors simply 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 application programs 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 controller is often hand-tweaked in assembly language, to squeeze every byte out of the ROM, and to use every register of RAM.

Other processors that are used in small devices include derivatives of the Motorola 68000 series [7]. For example, the Palm Pilots and their clones are based on the Motorola DragonBall CPUs (MC68328 [8]), and some cell phones are based on the same architecture family. These processors have maskable, prioritized, 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.

We are interested in providing tool support for programming interrupt-oriented small devices. In this paper we concentrate on the Z86 processor. Compared with the Motorola 68000, the Z86 has a much smaller instruction set and fewer interrupts (6 interrupts versus 18 in the case of DragonBall MC68EZ328). Yet the Z86 retains the powerful capabilities of vectored interrupt handling, making it attractive for rapid prototyping of programming tools.

We have designed and implemented a tool that supports three tasks that usually consume a significant part of a Z86-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. Our tool gives upper and lower bounds on the maximum stack size.

- **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. We use an implicit type system with just four types: interrupt information, code address, interrupt mask, unknown. Our tool checks that the data on top of the stack has the right type at the appropriate time.
- **Interrupt-Latency Analysis:** We study microcontrollers that need to handle interrupts within hard real-time bounds. Our tool gives upper bounds on the latencies.

We work with the bare Z86 assembly code.

Our tool is based on a known algorithm for model checking of pushdown systems [14]. That algorithm is closely related to the style of interprocedural analysis for C that has been studied by Reps and others [15]. The analysis of interrupts creates additional challenges, as we explain next.

The Problem

Given an assembly program, we first build a flow graph, and then run the desired analyses on the flow graph. The key question concerns the way we abstract a Z86-machine state into a node in the flow graph:

> *How much of a Z86-machine state should be represented in a flow-graph 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 our 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 [15], 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 we do 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.

Another consequence of not modeling the IMR is that for almost every node in a flow graph, it cannot be guaranteed that if an interrupt arrives at that point, then the interrupt will be handled within a finite amount of time. The core of the problem is that if a node has one outgoing edge to the interrupt handler and one or more other outgoing edges, then we cannot assume that the interrupt is enabled; we cannot conclude that the interrupt handler will be called at that point. This will be true of all such nodes, and we can therefore never conclude that an interrupt is certain to be handled.

The above observation makes it clear that we need to model at least some of the IMR. The IMR consists of seven bits, of which one is the "masterbit" which enables or disables all interrupt processing, and six others enable or disable individual interrupts. An interrupt will only be handled if both the masterbit and its "own" bit are set. When an interrupt handler is called, the masterbit 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 masterbit of the IMR. However, this is just as useless as modeling only the PC, as one of the tasks of an interrupt handler often is to re-enable interrupts by turning on the masterbit. 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 masterbit, leading to a stack growing without bounds, as above.

Our choice is to model 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 interrupts, each node in the flow graph can have up to six edges going to interrupt handlers, and one or more edges corresponding to normal 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; however, we did not investigate this idea.

One can imagine modeling *more* than the PC and the IMR, although it is not clear, in general, which other registers it would be beneficial to model. The key question is now:

> *Can modeling just the PC and the IMR lead to 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? Our criteria for usefulness are given by

- the degree to which the flow graph is a good basis for the three kinds of checks that we want a tool to 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 flow graph and do the checks.

The goal of this paper is to present an experimental evaluation of the above question.

Our Results

¿From a Z86-assembly program, our tool builds a flow graph in which each node represents the PC and the IMR. On that flow graph, we do the three desired kinds of checks. We have also built a software simulator of the Z86 and used it to find lower bounds on the maximum stack sizes. After experiments with seven proprietary microcontrollers, our results are the following.

- **Stack-Size Analysis:** For six of the seven benchmark programs, our tool gives an excellent estimate of the maximum stack size which was either exact (that is, equal to the lower bound that we found via simulation), or at most two more than the lower bound. For the seventh program, the stack size cannot be bounded without a more detailed analysis.

- **Type Checking of Stack Elements:** Our tool successfully checks the types of the stack elements for six of the seven benchmark programs.

- **Interrupt-Latency Analysis:** For the benchmark programs, our tool produced finite latencies for up to 79 % of the nodes in the control-flow graphs. For the remaining nodes, the tool found that 1) there were indeed paths from those nodes to the interrupt handlers (that is, the handler was live everywhere), but 2) there were cycles (potentially infinite loops) in the graph that made it impossible for the analysis to guarantee finite latencies.

Our tool is written in Java, it checks a Z86-assembly program in a few seconds, and it typically uses 30–40 MB of space, including the Java virtual machine.

Our tool is sufficiently fast and precise to be useful in practice. It enables increased confidence in correctness, it should significantly reduce testing requirements, and can support maintenance throughout the system life-cycle.

Rest of the paper: In the following section we give a small example of an interrupt-driven program and its flow graph. In Section 3 we describe the algorithms that we use, and in Section 4 we present our experimental results. Finally, in Section 5 we evaluate the prospects for scaling up our approach to processors in the Motorola 68000 family.

2 EXAMPLE

This section gives an informal presentation of concepts that will be rigorously defined in Section 3. Figure 1 shows a small Z86 program featuring a main program loop, and a single interrupt handler, both of which can call a shared procedure. Figure 2 shows the corresponding flow graph.

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, we have simplified the IMR representation 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.)

```
; 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.

  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              ; Musn'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 1: Example Program

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 2, 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.

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 inter-

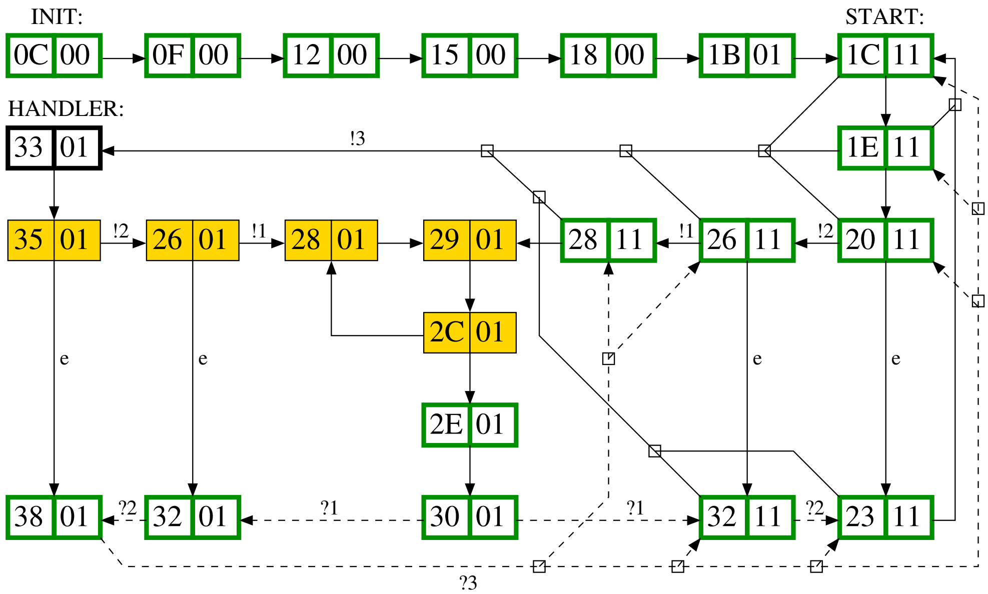


Figure 2: Example Program Flow Graph

rupt pushes three. Pop edges are distinguished with dashed lines. Additional "summary" edges generated by our analysis are labeled "e", 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. 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 rectangular nodes in the flow graph have a finite worst-case path to reach the interrupt handler. Rounded nodes, however, defy our current analysis of maximum interrupt latency. We call these nodes, "yellow nodes," as opposed to their "green," rectangular counterparts, who can definitely "go" to the interrupt in a finite amount of time.

The longest latency occurs during initialization, stretching from the node (0C,00) to the node (1C,11) where interrupts become enabled. The second-longest latency occurs when an interrupt arrives during execution of instruction 2E from within the interrupt handler. In this case, there is a delay of four instructions, as control passes through nodes (30,01), (32,01) and (38,01), before returning to a node with interrupts re-enabled.

In this particular example, the spate of yellow nodes in the central portion of the graph is the result of a possibly infinite loop between instructions 28, 29, and 2C. Because interrupts are masked off during this loop, our current analysis cannot determine an upper bound on the interrupt latency from this point. Furthermore, it is not possible to determine the maximum latency for nodes (35,01) or (26,01), because these nodes lead up to the cycle.

A casual examination of the actual code at line 2C reveals that this loop depends on decrementing an 8-bit register – clearly, this loop cannot continue for more than 255 repetitions. However, our current analysis cannot see this, because nodes in the graph do not carry the necessary additional state information. In future, we hope to be able to use checkable code annotations to establish bounds on data-dependent loops.

3 MODEL CHECKING

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 this does not come into play for our analysis.

The Z86 architecture supports an *indirect* register addressing mode. Our analysis relies on the unchecked assumption

instruction format	edge label	computation step
⟨various⟩	e	no change to the stack
PUSH IMR	!1 (IMR value)	the value of the IMR is placed on the stack
PUSH ⟨not IMR⟩	!1 “unk”	some value (not the IMR) is placed on the stack
CALL ⟨label⟩	!2 (return address)	procedure call
⟨interrupt call⟩	!3 (status register, return address)	implicit interrupt call
POP IMR	?1	the IMR is assigned the value on the top of the the stack
POP ⟨not IMR⟩	?1 “unk”	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 3: Instructions and the corresponding edge labels

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.

We only allow certain *direct* ways of manipulating the IMR, IRQ, and SP registers, as discussed below. Other forms of use can be located easily, and are checked by our tool.

IMR: We only allow IMR values 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.

IRQ: We assume that the IRQ is read only.

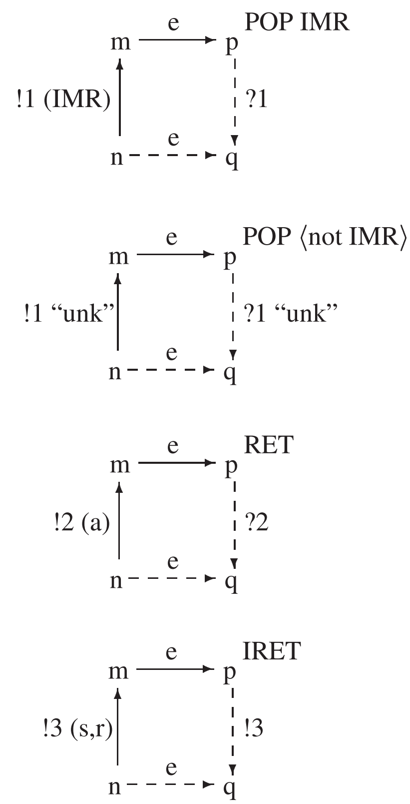
SP: We only allow the SP to be manipulated implicitly by stack-specific instructions or by an initialization instruction.

From Z86 Assembly Code to a Flow Graph

Given a Z86 assembly program, we build a flow graph in which each node is labeled with a PC value and an IMR value. The start node is labeled by 1) the PC value for the first line of the program, and 2) the IMR value where all bits are 0. We build the graph in a demand-driven way such that we only represent nodes that are reachable from the start node. 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.

There are nine kinds of edges, each with a distinctive label, as shown in Figure 3. 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” has weight 0, an edge with label “!n …” has weight n, and an edge with label “?n …” has weight $-n$. We use “unk” 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”. One kind of edge does not correspond to any instruction: the edges for implicit interrupt calls.

Conceptually, we insert edges in two steps. First we insert edges for implicit interrupt calls, and for instructions that place values on the stack or cause no change to the stack. Then we close the graph under the following four rules and the rule that the edges labeled “e” 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 above concern the generation of

1. *pop edges* that correspond to removing values from the stack, and
2. *summary edges* with label “e” that connect the point where values are placed on the stack to the point where the same values have just been removed again.

For example, let us consider the first rule in detail. The node

n is for an instruction "PUSH IMR", and there is an edge from n to m that models that the IMR is 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". 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. This edge represents removing the IMR value from the stack and assigning it to the IMR register. We also generate an edge (a summary edge) labeled "e" from n to q. This edge reflects that the stack size is the same at n and q, so it is warranted to allow a shortcut.

In Figure 2, the edges labeled "e" are summary edges, while we have left all other e-edges unlabeled for clarity.

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

Our algorithm can be understood as a demand-driven version of an algorithm for model checking of pushdown systems [14]. Unlike [14], our 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 [4].

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 [14]. We 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 we set the stack-size estimate to "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.

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. We use an implicit type system with just four types:

type ::= !1 "IMR" | !1 "unk" | !2 | !3.

We can map an edge label to a type in the obvious way.

We perform the check by ensuring that for every path of the form

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

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

$$n \xrightarrow{\text{!1 (IMR)}} m \overset{e}{\to} p \quad \text{and } p \text{ models "POP IMR"}$$
$$n \xrightarrow{\text{!1 "unk"}} m \overset{e}{\to} p \quad \text{and } p \text{ models "POP ⟨not IMR⟩"}$$
$$n \xrightarrow{\text{!2 (a)}} m \overset{e}{\to} p \quad \text{and } p \text{ models "RET"}$$
$$n \xrightarrow{\text{!3 (r,a)}} m \overset{e}{\to} p \quad \text{and } p \text{ models "IRET".}$$

Such checks correspond to the safety checks of Palsberg and Schwartzbach [13, 12], and can be implemented efficiently.

Interrupt-Latency Analysis

We concentrate on estimation of interrupt latency for the highest-priority interrupt (HPI). Other interrupts can easily be starved by the HPI, because as long as the HPI is enabled, it may fire all the time. We use $latency(n)$ to denote the latency at a node n, that is, the longest time it can take from when an HPI arrives at node n until its handler has been invoked. Our goal is to provide a conservative estimate of $latency(n)$ for all nodes n.

To simplify the explanations below, we assume that control can only be transferred to the start address of an interrupt handler when an interrupt is to be handled. Thus, we disallow all other forms of jumps to the start address of the handler. Our implementation does not rely on this assumption. Instead it uses the edge labels to distinguish between interrupt edges and noninterrupt edges.

We begin with coloring every node with one of the colors red, yellow, or green. The idea is that if the HPI arrives at a red node, then it will never be handled; if it arrives at a yellow node, then it may or may not be handled; and if it arrives at a green node, then it is guaranteed that it will be handled.

Let H be a predicate which is true of a node n if and only if the PC component of n is the start address of the handler for the HPI. We can think of H as the characteristic function for a set of nodes. We can now state a little more precisely the intuition behind the three colors. Informally, the color of a node is given by the following conditions:

$$\begin{array}{rcl} Red & \equiv & \text{it is not possible to reach a node in } \mathsf{H} \\ Yellow & \equiv & \neg(Red \vee Green) \\ Green & \equiv & \text{it is inevitable that computation} \\ & & \text{will reach a node in } \mathsf{H}. \end{array}$$

Let us now look in more detail at the circumstances under which computation will reach a node in H.

> **Key Observation:** If an HPI is pending when computation reaches a node n, and there is an edge from n to a node in H, then computation will proceed along such an edge.

The observation relies on modeling the IMR in full. At every node in the flow graph, the IMR value is exactly the value of the IMR in the underlying computation. Thus, an edge from n to a node in H indicates that the HPI is enabled at n, and therefore an HPI pending at n will be handled in the next computation step. We can use the observation to refine the informal definition of green nodes. We will say that a node is "ultragreen" if it has an edge to a node in H. The above observation entails that an ultragreen node is green.

$$\begin{array}{rcl} UltraGreen & \equiv & \text{there is an edge to a node in } \mathsf{H}. \\ Green & \equiv & \text{a node in } \mathsf{H}\text{, or} \\ & & \text{it is inevitable that computation} \\ & & \text{will reach an ultragreen node.} \end{array}$$

In Computation Tree Logic (CTL) [1] we can make the intuition precise by defining the colors as predicates on nodes:

$$\begin{array}{rcl} Red & \equiv & \neg\, \mathsf{EF}(\mathsf{H}) \\ Yellow & \equiv & \neg(Red \vee Green) \\ UltraGreen & \equiv & \mathsf{EX}(\mathsf{H}) \\ Green & \equiv & \mathsf{H} \vee \mathsf{AF}(UltraGreen). \end{array}$$

It is straightforward to check that a node has exactly one of the colors Red, $Yellow$, $Green$, and that an $UltraGreen$ node is $Green$.

Given a flow graph G and a formula φ in CTL, it can be decided in $O(|G| \times |\varphi|)$ time whether φ is true or false of the nodes in G. Our implementation is specialized to the particular formulas above, and may be faster than a general implementation of a model checker for CTL.

For a red node n, the latency at n is infinite; for a yellow node n, we conservatively estimate the latency at n to be infinite; and for a green node n, we calculate an upper bound on the latency at n as follows.

We first annotate each edge in the flow graph with the time it takes for the CPU to execute the underlying computation. For a path p, we use $time(p)$ to denote the sum of the timings of the edges on p. We use $ictime$ ("interrupt call time") to denote the time it takes to execute an interrupt call. Notice that $ictime$ is the timing of any interrupt edge to a node in H. We use $P(n)$ to denote the set of paths from n to an ultragreen node (where that ultragreen node is the first ultragreen node on the path). We then calculate an estimate of $latency(n)$ as follows:

$$maxtime(n) = \max_{p \in P(n)} time(p)$$

$$latencyestimate(n) = \begin{cases} \text{infinite} & \text{if } Red(n) \vee Yellow(n) \\ 0 & \text{if } \mathsf{H}(n) \\ maxtime(n) + ictime & \text{if } (\mathsf{AF}(UltraGreen))(n) \end{cases}$$

For a green node n not in H (that is, $(\mathsf{AF}(UltraGreen))(n)$), the latency estimate is the maximum time it takes to get from n to an ultragreen node plus the time it takes to execute an interrupt call.

Ideally, the graph would contain only green nodes, and we would then give a finite upper bound on the HPI-latency across the whole program by taking the maximum over all nodes. This upper bound for the whole program is a conservative measure of the responsiveness of the microcontroller.

4 EXPERIMENTAL RESULTS

Framework and Benchmarks

All algorithms were implemented in Java, and run on the IBM Java2 SDK 1.3 [3]. Runs were made on a 500 MHz Pentium3-based laptop.

Our implementation has been tweaked for speed, but there is still room for improvement. Space usage has not really been optimized, so it 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 most 40 MB) for our experiments. 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 harddisk speed or cache behavior from influencing the 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).

For our experiments, we have used seven proprietary microcontroller systems provided by Greenhill Manufacturing, Ltd. (http://www.greenhillmfg.com/). Greenhill has over a decade of experience producing environmental controls systems for agricultural needs. 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.

We have also experimented with a 60-line example program. It is a small, representative piece of code, which we believe embodies the most difficult interrupt-driven behavior present in the seven proprietary programs. It is important to note that the example program is more daring with its interrupt behavior than many of the production-run programs; the designer has indicated to us that the example program exemplifies the responsiveness that he would like to have in the other seven programs, but was unwilling to try without more comprehensive guarantees of correct stack behavior.

Building the graph

The size of the graph is illustrated by the number of *nodes* and *edges*. The elements represent the possible top of the stack elements stored in each node. We report the number of elements stored in the entire graph. Space is in megabytes.

Only half of the time and space is spent building the graph. The rest is used to start the virtual machine and parse the assembler file. Our parser uses `JavaCC` [5] and `JTB` [16].

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
Example	148	222	3.16 s	34.9 MB

Stack-Size Analysis
The upper bounds on the stack sizes found by the analysis are reported in the following table, in the column Upper. Our stack-size analysis typically takes around 0.1 seconds, and it takes little extra memory. Notice that the columns Time and Space include the cost of building the graph.

Stack-size analysis				
Program	Lower	Upper	Time	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
Example	37	37	3.21 s	34.9 MB

Our current analysis is unable to ascertain an upper bound on the `Fan` program because it has the assembler equivalent of a `for` loop with a PUSH in the body. While it is obvious to a programmer that the number of loop iterations (and therefore the stack size) is bounded, our algorithm cannot see this based solely on the PC and IMR registers. Extending our analysis to handle the `Fan` program is a chief goal of our future work.

The upper bounds presented above might not have corresponded to paths actually realizable in the running microcontroller programs. To evaluate the precision of the upper bounds, we used the following approach to find lower bounds in actual program runs.

We have implemented a simulator for the Z86E30 microcontroller, except for a few obscure features that are not used by our benchmark programs. We also simulate the external devices with which our benchmark programs communicate. These devices include an intelligent display, an A/D converter, a serial port, and an EEPROM chip. Our simulator monitors the stack size, and records the maximal value together with the corresponding program path. With the simulator, we can find a lower bound on the maximal stack size. Any run of the program with some interrupt schedule will generate such a lower bound.

The simulator is implemented in Java and runs on IBM's Java2 SDK 1.3 with JIT. On a 500 MHz Pentium3 based laptop the simulator runs 4 times faster than the real hardware with a 8 MHz clock.

The input to the simulator is an assembly program and an interrupt schedule. The schedule consists of a number of interrupt sequences that should be fired during the run in order to test the assembly program. Our tool supports single-point interrupts and periodic interrupts.

We use several strategies to search for an interrupt schedule that gives as tight a lower bound as possible. We tried 1) a schedule written by a person familiar with the microcontroller, 2) 1,000 randomized schedules, and 3) 1,000 schedules generated by a genetic algorithm. The genetic algorithm consistently matched or outperformed the results of the other two approaches. The lower bounds that we found are reported in the table above, in the column Lower.

Type Checking of Stack Elements
For the six benchmark programs for which our tool produced a finite stack size, all stack operations type check. Our tool carries out the checks while executing the closure rules that insert pop edges.

Our notion of 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."

Interrupt-Latency Analysis
The results of our interrupt-latency analysis are shown in Figure 4.

In all cases, the number of Red nodes is listed as 0%; this reflects that there were *no* red nodes in any of the graphs.

The *latency* is the maximum number of machine cycles from any of the green nodes to one of the nodes corresponding to the interrupt handler. In other words, it is the longest interrupt latency in the part of the graph that we can analyze. Our tool can give the actual execution path that gives the maximum latency, allowing the programmer to reconsider latency bottlenecks.

Our reported time and space results do not include calculating, storing and printing the maximum latency path from each green node. However, the extra space and time requirement is trivial.

The analysis itself takes 0.8 - 1.6 seconds in most cases, and takes up almost no extra memory beyond what is used to store the graph. The `Fan` program is an exception, being the largest program, with the largest number of active interrupt handlers.

For all the benchmarks, the maximum latency is at most 326 machine cycles, (that is, 0.326 milliseconds on this processor), for the portion of the graph that we can analyze. The

Interrupt latency analysis of highest priority IRQ								
Program	Ultragreen	Green	Ultrayellow	Yellow	Red	Latency	Time	Space
CTurk	43%	51%	34%	49%	0%	260	4.82 s	31.6 MB
GTurk	43%	50%	30%	50%	0%	272	5.79 s	32.2 MB
ZTurk	42%	50%	30%	50%	0%	276	5.45 s	32.1 MB
DRop	15%	19%	60%	81%	0%	312	4.76 s	31.2 MB
Rop	15%	19%	58%	81%	0%	312	4.94 s	31.8 MB
Fan	56%	67%	24%	33%	0%	310	44.44 s	39.6 MB
Serial	43%	79%	14%	21%	0%	326	4.00 s	31.0 MB
Example	25%	46%	30%	54%	0%	242	3.24 s	34.9 MB

Figure 4: Interrupt-latency analysis

326 machines cycles include the 20 cycles it takes to execute an interrupt call.

Figure 4 lists the percentages of both yellow nodes and so-called ultrayellow nodes. Every ultrayellow node is yellow. An ultrayellow node is one that occurs on a cycle of only yellow nodes. In other words, every cycle of yellow nodes is also a cycle of ultrayellow nodes. All other yellow nodes are on a path to such a cycle. As shown in Figure 4, the percentage of ultrayellow nodes is considerably smaller than the percentage of yellow nodes. So, if an ultrayellow cycle can be converted to a green cycle, other yellow nodes will be automatically converted to green as well.

5 CONCLUSION

Our experiments 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.

Our tool 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 insurmountable 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 [9], and loop-invariant specification [10, 11] are successful techniques that it may be possible to combine with our analysis to tackle the upper bounds on the seventh program.

As for calculating maximum interrupt latency, further work must be done to reduce the number of yellow nodes in the graph before our algorithm will be able to provide truly useful results. At present, only a small percentage of nodes in the graph are green, but not ultragreen. In other words, our analysis can find a finite, nontrivial latency bound for relatively few of the nodes in most of the programs. All the remaining nodes either have no latency whatsoever (ultragreen,) or cannot be bounded with our current tool, (yellow).

Reduction of yellow nodes goes hand in hand with deeper analysis of loop invariants, as most of the yellow nodes are caused by data-dependent loops that our tool cannot presently see will terminate.

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. We believe that with additional work, our analysis could scale up to programs written for Palm Pilots, handheld digital phones, and many other interrupt-oriented applications. While our Z86 programs 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 our 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 our analysis, and only add details to the complexity of our implementation.

The developers of PalmOS (the operating system running on the Palm Pilots and clones) are concerned with interrupt latency. In the HandSprings Developer Technical Information [2] they give the latencies for a slightly modified version of PalmOS. With a tool such as ours, one can hope to produce upper bounds on latencies for their system, and to identify code bottlenecks where improvements would help most.

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. 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 jumped address could cause computation to stop, due to a bus error, a protection error, or a misaligned memory address. For these reasons, we believe that Typed Assembly Language [6] could provide the necessary structure to guarantee program progress in such a scaled-up framework. As an added bonus, typing annotations may assist in eliminating yellow nodes in the graph, 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 we have been forced to neglect in our current analysis.

Our tool is one of the first to give an efficient and useful static analysis of assembly code. A recent tool [17] addresses a complementary set of problems; it analyzes SPARC machine code with the purpose of enforcing safety policies for resource access. It remains to be seen if the two approaches can be combined.

Acknowledgments: We thank Greenhill Manufacturing, Ltd., for the use of their proprietary software as test subjects in our experiments. We thank Andreas Podelski for suggesting the use of model checking for pushdown systems, and we thank Greg Frederickson for helpful discussions about algorithmic issues. Palsberg was supported by a National Science Foundation Faculty Early Career Development Award, CCR–9734265, and by CERIAS (Center for Education and Research in Information Assurance and Security).

REFERENCES

[1] 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*, volume B, pages 995–1072. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[2] Handspring. *Handspring Developer Technical Information Technical FAQ*, 2000. http://www.handspring.com/developers/tech_faq.asp#q13.

[3] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a Java just-in-time compiler. In *Proceedings of the Fifteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, 2000.

[4] David McAllester. On the complexity analysis of static analyses. In *Proceedings of SAS'99, 6th International Static Analysis Symposium*, pages 312–329. Springer-Verlag (*LNCS* 1694), 1999.

[5] Sun Microsystems. The Java compiler compiler. www.suntest.com/Jack/, 1997.

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

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

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

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

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

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

[12] Jens Palsberg and Patrick M. O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995. Preliminary version in Proceedings of POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 367–378, San Francisco, California, January 1995.

[13] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995.

[14] Andreas Podelski. Model checking as constraint solving. In *Proceedings of SAS'00, International Static Analysis Symposium*, pages 22–37. Springer-Verlag (*LNCS* 1824), 2000.

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

[16] Kevin Tao and Jens Palsberg. The Java tree builder. Purdue University, www.cs.purdue.edu/jtb, 1997.

[17] Zhichen Xu, Barton P. Miller, and Thomas Reps. Safety checking of machine code. In *Proceedings of ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 70–82, 2000.

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