Eliminating Stack Overflow by Abstract Interpretation

JOHN REGEHR, ALASTAIR REID, and KIRK WEBB
University of Utah

An important correctness criterion for software running on embedded microcontrollers is *stack safety*: a guarantee that the call stack does not overflow. Our first contribution is a method for statically guaranteeing stack safety of interrupt-driven embedded software using an approach based on context-sensitive dataflow analysis of object code. We have implemented a prototype stack analysis tool that targets software for Atmel AVR microcontrollers and tested it on embedded applications compiled from up to 30,000 lines of C. We experimentally validate the accuracy of the tool, which runs in under 10 sec on the largest programs that we tested. The second contribution of this paper is the development of two novel ways to reduce stack memory requirements of embedded software.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and Embedded Systems; D.2.4 [**Software/Program Verification**]: Formal Methods; Reliability; Validation; D.3.3 [**Language Constructs and Features**]: Concurrent Programming Structures; F.3.2 [**Semantics of Programming Languages**]: Program Analysis; D.3.4 [**Processors**]: Memory Management; D.3.3 [**Language Constructs and Features**]: Dynamic Storage Management

General Terms: Languages, Reliability, Verification

Additional Key Words and Phrases: Microcontroller, call stack, interrupt-driven, abstract interpretation, dataflow analysis, context sensitive, sensor network

1. INTRODUCTION

Inexpensive microcontrollers are used in a wide variety of embedded applications such as vehicle control, consumer electronics, medical automation, and sensor networks. Static analysis of software running on these processors is important for two main reasons. First, embedded systems are often used in safety critical applications and can be hard to upgrade once deployed. Since undetected bugs can be very costly, it is useful to attempt to find software defects early. Second, severe constraints on cost, size, and power make it undesirable to overprovision resources as a hedge against unforeseen demand. Rather, worst-case resource requirements should be determined statically and accurately,

Authors' address: University of Utah, Salt Lake City, UT 84112.

even for resources like memory that are convenient to allocate in a dynamic style.

In this paper, we describe the results of an experiment in applying static analysis and transformation techniques to embedded software in order to bound and reduce its stack memory requirements. We check software for *global stack safety*: the property that it will not run out of stack memory at run time. Stack safety, which is not guaranteed by traditional type-safe languages like Java, is particularly important for embedded software because stack overflows cause memory corruption that can easily crash a system or otherwise lead to incorrect operation. The transparent dynamic stack expansion that is performed by general-purpose operating systems is infeasible on small embedded systems due to their small physical memories and lack of virtual memory hardware. For example, 8-bit microcontrollers typically have between a few tens of bytes and a few tens of kilobytes of RAM. Bounds on stack depth can also be usefully incorporated into programs, for example, to assign appropriate stack sizes to threads or to provide a heap allocator with as much storage as possible without compromising stack safety.

The focus of our work is practical: the primary goal is to develop techniques that can be implemented in tools that are useful for people developing embedded software. In several cases, we had to abandon sound, but pessimistic, analysis techniques in favor of unsound techniques that produce better results while placing added validation burden on developers. We attempt to provide a frank discussion of the tradeoffs between sound and unsound techniques.

During the course of this project, we developed a prototype tool that analyzes an executable embedded program in two passes. First, it performs context-sensitive dataflow analysis in order to identify unexecutable branches and to estimate the set of possible preemption relations between interrupt handlers. Second, it puts together worst-case stack depth results for individual interrupt handlers to compute a global worst-case stack depth. Much of our work has focused on generating tight bounds for interrupt-driven embedded software. Our tool successfully estimates the maximum stack depth of most of the programs shipped with TinyOS [Hill et al. 2000], an operating system for sensor network nodes based on Atmel's AVR architecture.

1.1 Stack Safety by Testing

Static analysis is a relatively new approach to achieving stack safety; its relationship with the more established testing-based approach is illustrated in Figure 1. An application note for Texas Instruments digital signal processors (DSPs) [Alter 2003] provides a good summary of what can be accomplished through testing:

> A stack overflow in an embedded DSP application generally produces a catastrophic software crash due to data corruption, lost return addresses, or both. The traditional approach to avoiding stack overflow is to perform offline testing during software development. Typically, a stack will be comfortably oversized, and . . . the application

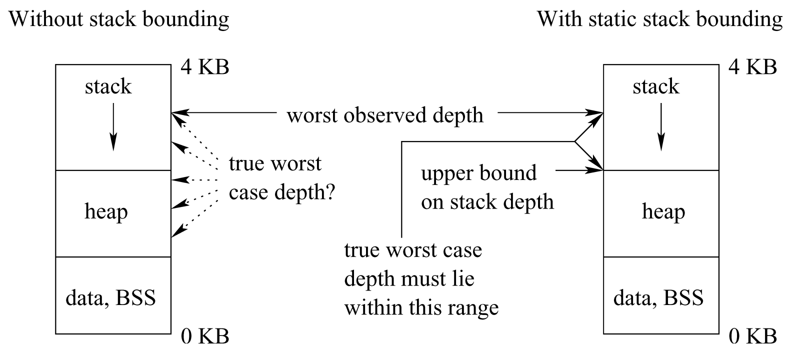

Without stack bounding
With static stack bounding
stack
heap
data, BSS
4 KB
0 KB
worst observed depth
true worst case depth?
upper bound on stack depth
true worst case depth must lie within this range

Fig. 1. Typical RAM layout for an embedded program with and without stack bounding. Without a bound, developers must rely on guesswork to determine the amount of storage to allocate to the stack.

> software will then be run over some period of time (e.g., hours, days, or sometimes even weeks). . . . While this offline method of stack sizing is invaluable as a first pass approach, it does not eliminate the possibility of a stack overflow occurring at runtime. Programmers may therefore use a larger stack than they might actually need, which can waste valuable on-chip RAM resources.

The main problem with testing-based approaches to software validation is that typically some executable paths through the system are missed. Worst-case stack depth of interrupt-driven code is particularly difficult to discover through testing because it depends on what code is executing when each interrupt is triggered, and on whether further interrupts fire before the first returns. For example, consider a hypothetical embedded system where the maximum stack depth occurs when the following events occur at almost the same time: (1) the main program summarizes data once a second spending 100 μs at maximum stack depth; (2) a timer interrupt fires 100 times a second, while the handler spends 100 μs at maximum stack depth; and (3) a packet arrives on a network interface up to 10 times a second and the network interrupt handler spends 100 μs at maximum stack depth. If these events occur independently of each other, the worst case will occur roughly once every 32 years. This means that the worst case will probably not be discovered during testing, but that if just 10,000 instances of the system are deployed, we would expect the maximum stack depth to be reached every day.

A final drawback of the testing-based approach to determining stack depth is that it treats the system as a black box, providing developers with little or no feedback about how to best optimize memory usage.

1.2 Tool Support for Bounding and Reducing Stack Size

Our goal was to develop techniques that could be embodied in a tool that would be useful for embedded developers, eliminating or greatly reducing the amount of testing required to show stack safety. We wanted the tool to have the following properties:

- Conservative: the tool should never underestimate the true worst-case stack depth.
- Precise: the stack depth bound should be as small as possible without being inaccurate.
- Fast: the tool should be usable interactively, as opposed to being run overnight.
- Usable: the tool should insulate the developer from details of the underlying static analysis for example, by providing good error messages when it fails.
- Informative: the tool should alert developers to potential unsoundnesses in the analysis and it should be able to provide useful information about a system, such as, the stack usage of each function, the path through the system that produces the worst-case stack depth, etc. This information can help developers identify good candidates for manual storage optimizations.

Our stack analysis tool is a work in progress, but we believe that it has been reasonably successful at meeting all of our goals. Perhaps the most interesting and surprising thing that we learned during the course of this research is that many embedded systems have a very static underlying structure of interrupt preemption relations and that this structure can be recovered using static analysis. In fact, for many of the embedded systems that we looked at, our stack tool can identify a static interrupt mask for more than 99% of instructions in all analysis contexts.

Using our method for statically bounding stack depth as a starting point, we have developed two novel ways to reduce the stack memory requirement of an embedded system. The first optimization is completely automatic; it evaluates the effect of a large number of potential program transformations in a feedback loop, applying only transformations that reduce the worst-case depth of the stack. Static analysis makes this kind of optimization feasible by rapidly providing accurate information about a program. The second optimization is not entirely automatic—it requires guidance from developers to avoid unsafe transformations. It works by eliminating unnecessary preemption relations between interrupt handlers, often leading to reduced stack memory requirements.

Our work is preceded by a whole-program analysis for bounding the stack depth of Z86 binaries by Brylow et al. [2001]. While they focused on relatively small programs written by hand in assembly language, we focus on programs that are up to 30 times larger and that are compiled from C to a RISC architecture. The added difficulties in analyzing larger, compiled programs required us to develop a more powerful dataflow analysis based on context-sensitive abstract interpretation of machine code and also to treat the dataflow analysis and stack-depth analysis separately.

The rest of this paper is organized as follows. Section 2 motivates our approach to bounding stack depth and introduces a key abstraction: the interrupt preemption graph (IPG). In Section 3 we describe the dataflow analysis that supports construction of the IPG, and Section 4 presents the analysis that computes stack depth using an IPG as input. Section 5 evaluates our prototype stack tool and describes various ways in which we validated its output. In Section 6, we describe two novel ways to reduce worst-case stack memory requirements. Finally, we compare our research to previous efforts in Section 7 and conclude in Section 8.

2. APPROACH

Typically, embedded system designers statically allocate resources whenever possible. This makes systems more predictable and reliable by providing a priori bounds on resource consumption. However, an almost universal exception to this rule is that memory is dynamically allocated on the call stack. Stacks provide a useful model of storage, with constant-time allocation and deallocation and without fragmentation. Furthermore, the notion of a stack is designed into microcontrollers at a fundamental level. For example, hardware support for interrupts typically pushes some or all of the processor state onto the stack before calling a user-defined interrupt handler and pops the machine state upon termination of the handler.

Our prototype stack analysis tool targets executable programs for the Atmel AVR, a popular family of microcontrollers. We chose to analyze object code, rather than source code, for a number of reasons:

- There is no need to predict compiler behavior. Many compiler decisions, such as those regarding function inlining and register allocation, have a strong effect on stack depth.
- Inlined assembly language is common in embedded systems, making source-code analysis difficult: it requires knowledge of the high-level language semantics, the assembly language semantics, and also the (typically poorly documented) interface between the two.
- The source code for libraries and real-time operating systems is commonly not available.
- Since object code analysis is independent of the compiler, developers are free to change compilers or compiler versions. In addition, the analysis is not fragile with respect to nonstandard language extensions that embedded compilers use to provide developers with fine-grained control over processor-specific features.
- Adding a postcompilation analysis step to the development process presents developers with a clean usage model: since we examine the (usually singular) end-product rather than multiple input files, there is no need to modify the makefiles or hook into the development environment.

Analyzing the stack memory requirements of sequential code is straightforward, with a few important exceptions that we return to later. The more difficult challenge in embedded systems is accurately bounding stack usage of interrupt-driven code. By using dedicated hardware to detect events, interrupts reduce the amount of polling that needs to be performed in software, enabling important performance optimizations for resource-poor embedded processors. For example, many processors have the ability to sleep while waiting for an interrupt to arrive, reducing power consumption by a factor of 1000 or more [Hill et al. 2000].

An interrupt can fire any time it is *enabled* and the processor has just finished executing an instruction. The criteria for enabling an interrupt vary across processor architectures, but typically there is a *master interrupt enable bit* that disables all interrupts when cleared, in addition to a number of enable bits that control individual interrupts. An interrupt is enabled when its enable bit and the master interrupt enable bit are both set.

Interrupts introduce concurrency into an embedded system: we say that interrupt handler X *preempts* interrupt handler Y, if X begins to run before Y completes. Interrupts are never blocked in the sense that threads are—they run to completion except when preempted. Trivially, every interrupt preempts the processor's main (noninterrupt) context. Temporary state used by an interrupt handler is pushed onto the call stack and, therefore, the maximum stack depth for a system is strongly dependent on interrupt behavior.

Consider an embedded system containing n interrupt handlers, where concurrent execution of interrupts is prohibited. This is trivially implementable by ensuring that all interrupt-mode code runs with the master interrupt enable bit cleared. In this case, a safe bound on stack depth can be computed as follows:

$$\text{stack bound} = \text{depth}(\text{main}) + \max_{i=1..n} \text{depth}(\text{interrupt}_i) \tag{1}$$

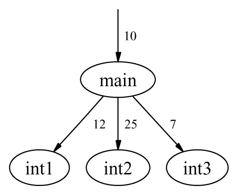
However, interrupt handlers are often run with interrupts enabled to ensure that other interrupt handlers are able to meet real-time deadlines. If a system permits, at most, one outstanding instance of each interrupt handler, the worst-case stack depth of a system can be computed using this formula:

$$\text{stack bound} = \text{depth}(\text{main}) + \sum_{i=1..n} \text{depth}(\text{interrupt}_i) \tag{2}$$

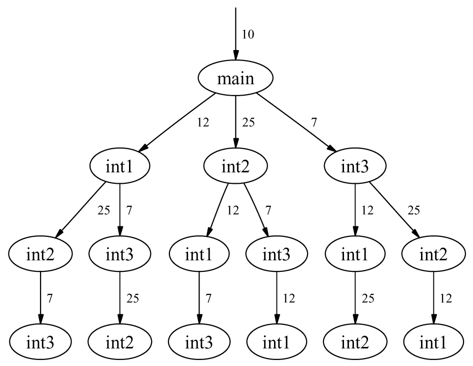
It is common for an embedded system to run some interrupt handlers with interrupts disabled and run some with interrupts enabled. This means that Eq. (1) is usually not applicable and Eq. (2) is overly conservative: it accounts for preemptions that can never happen, leading to a pessimistic estimate of stack memory requirements.

To obtain a safe, tight stack bound for realistic embedded systems, we developed a two-part analysis. The first is a context-sensitive dataflow analysis that identifies unexecutable branches, computes the state of the interrupt mask at each program point, and computes the worst-case stack memory requirements of sequential code: individual interrupt handlers; the main context. The second part of the analysis—unlike the first—accounts for potential preemptions between interrupts handlers. The results of the dataflow analysis are used to compute an *interrupt preemption graph* (IPG): a weighted, directed graph where each edge corresponds to a potential preemption by an interrupt handler; edge weights correspond to stack memory requirements. For example, Figure 2a shows an IPG corresponding to Eq. (1) and Figure 2b corresponds to Eq. (2). The global stack memory requirement for a system can be computed by searching for the longest path through the IPG.

The problem of computing the exact worst-case stack depth of a system is undecidable, as are most other static analysis problems. However, it can be approximated.



(a) Corresponds to Equation 1



(b) Corresponds to Equation 2

Fig. 2. Example interrupt preemption graphs.

```
in      r24, 0x3f    ; r24 <- CPU status register
cli                  ; disable interrupts
adc     r24, r24     ; carry bit <- prev interrupt status
eor     r24, r24     ; r24 <- 0
adc     r24, r24     ; r24 <- carry bit
mov     r18, r24     ; r18 <- r24

... critical section ...

and     r18, r18     ; test r18 for zero
breq    .+2          ; if zero, skip next instruction
sei                  ; enable interrupts
ret                  ; return from function
```

Fig. 3. This fragment of assembly language for Atmel AVR microcontrollers motivates our approach to program analysis and illustrates a common idiom in embedded software: disable interrupts, execute a critical section and then reenable interrupts only if they had previously been enabled.

3. DATAFLOW ANALYSIS

This section describes the first phase of our stack-depth analysis: a context-sensitive dataflow analysis of object code. Of course, dataflow analysis is a broadly useful technology that has many applications beyond stack depth analysis. In the future we plan to decouple our dataflow analyzer from the stack-depth analyzer in order to make it separately usable.

The first challenge in bounding stack depth is to measure the contributions to the stack of each interrupt handler and of the main program. Since indirect function calls and recursion are uncommon in embedded systems [Engblom 1999], a callgraph for each entry point into the program can be constructed using standard analysis techniques. Given a callgraph, it is usually straightforward to compute its stack requirement.

Figure 3 presents a fragment of machine code that motivates our approach to program analysis. Its purpose is to disable interrupts, execute a critical section that must run atomically with respect to interrupt handlers, and then reenable interrupts only if they had previously been enabled. Code analogous to this can be found in almost any embedded system. Analysis of this fragment is successful if the state of the master interrupt enable bit is known at the end of the code fragment whenever its state is known at the start. It is important not to lose track of the state of the interrupt mask because this prevents construction of a precise interrupt preemption graph.

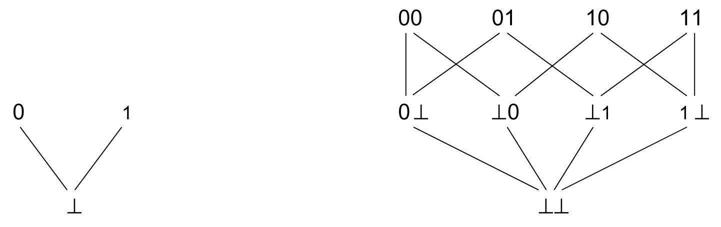
There are a number of challenges in analyzing such code. First, the effects of arithmetic and logical operations must be modeled with sufficient accuracy to track data movement through general-purpose and I/O registers. In addition, partially unknown data must be modeled. For example, analysis of the code fragment should succeed even when only a single bit of the CPU status register—the master interrupt control bit—is initially known. Second, dead edges in the control-flow graph must be detected and avoided. For example, when the example code fragment is called in a context where interrupts are disabled, it is important that the analysis conclude that the `sei` instruction is not executed, since this would pollute the estimate of the interrupt mask at subsequent addresses.

3.1 Abstracting the Machine State

Abstract interpretation [Cousot and Cousot 1977] provides a framework for program analyses such as our dataflow analysis. Two important design decisions for an abstract interpreter are: What part of the machine state should be modeled and what should be the model for each element of the machine state?

For most programs that we have analyzed, stack depth can be tightly bounded by modeling only the program counter, general-purpose registers, and several I/O registers. The Atmel AVR is an 8-bit architecture with 32 general-purpose registers and a variable number of I/O registers; the ATmega128 chips that are the basis of the mica2 sensor network nodes that we examine in Section 5 have 224 I/O registers. From the I/O space, we model the registers that contain interrupt masks, the processor status register, and the stack pointer. Our analyzer does not model main memory or most I/O registers, such as those that implement timers, analog-to-digital conversion, and serial communication. Several programs in our test suite can benefit from modeling values stored on the stack and we have added an experimental feature to our analyzer supporting this. We discuss this feature further in Section 3.5.

We chose to model each element of machine state at the bit level to capture the effect of bitwise operations on the interrupt mask and condition code register. (We had initially attempted to model the machine at word granularity, but this turned out to lose too much information through conservative approximation.) Each bit of machine state is modeled using the lattice depicted in Figure 4a. The lattice contains the values 0 and 1 as well as a bottom element, $\perp$, that corresponds to a bit that cannot be proved to have a concrete value. Unknown values are introduced into the analysis in three main ways. First, external input to a program, such as a value returned from a sensor, is truly unknown at analysis time; these values must be represented as vectors of $\perp$. Second,



(a) One bit (b) Two bits

Fig. 4. Bitwise lattices.

merge

	1	0	⊥
1	1	⊥	⊥
0	⊥	0	⊥
⊥	⊥	⊥	⊥

and

	1	0	⊥
1	1	0	⊥
0	0	0	0
⊥	⊥	0	⊥

or

	1	0	⊥
1	1	1	1
0	1	0	⊥
⊥	1	⊥	⊥

xor

	1	0	⊥
1	0	1	⊥
0	1	0	⊥
⊥	⊥	⊥	⊥

(a) Combining abstract bits at merge points

(b) Logical operations on abstract bits

Fig. 5. Operations on bitwise values.

since we do not model the complete machine state, when data is read from an unmodeled part of the system, the result is represented as vectors of $\perp$. Third, when control flow paths are merged, for example, after analyzing both branches of an if-then-else construct, the analysis must make a conservative approximation of the value of each bit in the result using the *merge* function illustrated in Figure 5a.

The lattice for a machine word is the composition of multiple single-bit lattices, as shown in Figure 4b. The first use of the bitwise domain for analyzing software that we are aware of is by Razdan and Smith [1994].

3.2 Machine-Level Operations in the Bitwise Domain

Figure 5b shows abstractions of some common logical operators. Abstract operators should always return a result that is as precise as possible. For example, when all bits of the input to an instruction have the value 0 or 1, the execution of the instruction should have the same result that it would have on a real processor. In this respect our abstract interpreter implements most of the functionality of a standard CPU simulator.

For example, when interpreting the and instruction with $\{1, 1, 0, 0, 1, 1, 0, 0\}$ as one argument and $\{\perp, \perp, \perp, \perp, 1, 1, 1, 1\}$ as the other argument, the result register will contain the value $\{\perp, \perp, 0, 0, 1, 1, 0, 0\}$. Arithmetic operators are treated similarly, but require more care because bits in the result typically depend on multiple bits in the input. Furthermore, the abstract interpretation must take into account the effect of instructions on processor condition codes, since subsequent branching decisions are made using these values.

The example in Figure 3 illustrates two commonly used special cases that must be accounted for in the abstract interpretation. First, the add-with-carry instruction adc, when both of its arguments are the same register, acts as rotate-left-through-carry. In other words, it shifts each bit in its input one position to the left, with the leftmost bit going into the CPU's carry flag and the previous carry flag going into the rightmost bit of the register. Second, the exclusive or instruction `eor`, when both of its arguments are the same register, acts like a clear instruction. After its execution, the register is known to contain all zero bits regardless of its previous contents.

We initially implemented an abstract version of each instruction by hand; this was time consuming and the resulting code was often imprecise and sometimes buggy. We tackled all three problems in subsequent work: the Hoist toolchain [Regehr and Reid 2004] permits us to automatically construct *maximally precise* abstract operations for the ALU operations in the AVR instruction set. An abstract operation is maximally precise if, for every input, it produces the most precise (greatest in the lattice sense) result. We accomplished this by exhaustively computing the most precise results for all abstract input values, summarizing the result in a binary decision diagram (BDD), and converting the BDD to efficient C code that could be linked into our analysis tool. A side benefit of automating this task was that it became easy to uniformly handle special cases, such as those where both arguments came from the same register.

3.3 Managing Abstract Processor States

An important design decision for an abstract interpreter is when to create a copy of the abstract machine state at a particular program point, as opposed to merging two abstract states. If states are merged too infrequently, the analysis will consume too many resources. On the other hand, if states are merged too often, the analysis will return excessively imprecise results, because merges lose information.

There are two common criteria for merging states: context-(in)sensitivity and path-(in)sensitivity. An analysis is *context-insensitive* if it merges states of all invocations of a given function. Conversely, it is *context-sensitive* if each invocation of the same function is analyzed separately. An analysis is *path-insensitive* if it merges states at join-points in the control-flow graph (e.g., at the end of an if-then-else statement or at the head of a loop). Conversely, it is *path-sensitive* if each path through a function is analyzed separately.

Our initial analyzer was path- and context-insensitive. It ran quickly but did not return acceptably precise results. For example, for many embedded codes that we analyzed, it was unable to find a concrete interrupt mask for more than 50% of instructions, leading to the construction of highly pessimistic interrupt preemption graphs. Our current tool is context sensitive. This is important, because, in many systems, code like that shown in Figure 3 is called with interrupts disabled by some parts of the system and is called with interrupts enabled by other parts of the system. The context-sensitive version of our tool is able to identify a constant interrupt mask for most instructions in most systems that we looked at.

In the worst case, a context sensitive analysis requires space and time exponential in the depth of the callgraph for a system. In practice we have not found this to be a problem: the largest programs that we have analyzed cause the analyzer to allocate about 215 MB. If memory requirements become a problem for the analysis, a relatively simple solution would be to merge program states that are identical or that are similar enough that a conservative merging will result in minimal loss of precision.

3.4 Analysis Algorithm

The abstract interpreter begins by initializing a worklist with all entry points into the program; entry points are found by examining the vector of interrupt handlers that is stored at the bottom of a program image, which includes the *reset vector*: the address of a startup routine that eventually jumps to `main()`. While the worklist is not empty, the analyzer removes an item and processes it. Each item corresponds to interpreting a single instruction. If interpreting an instruction changes the state of the processor at that program point, items are added to the worklist corresponding to each live control flow edge leaving the instruction. Termination is assured because the bitwise lattice has finite height and because the abstract operations are *monotone*—they always push the machine state toward the bottom of the lattice.

3.5 Assumptions, Limitations, and Challenges

An ideal static analysis tool would be *sound*, always returning a conservative approximation of the state of the machine at each program point. We found that there is a strong conflict between creating a sound tool and creating a useful tool. This section describes some soundness issues and other problems that we encountered while creating the stack depth analyzer; it also shows our solutions.

3.5.1 *Indirect Stores.* The AVR architecture maps the general purpose and I/O registers into memory space. A sound analysis would be forced to drop all knowledge of any register value every time it encountered an indirect store whose target was not provably outside of the register file. In practice this tool would be useless: it could not return tight stack bounds because real programs contain a large number of indirect stores with indeterminate targets. An industrial-strength stack tool would probably benefit from good alias analysis. However, alias analysis of object code [Debray et al. 1998; Balakrishnan and Reps 2004] is a difficult problem that is beyond the scope of our current work. We, therefore, assume that indirect stores do not modify registers.

Another problem caused by indirect stores is the potential for a program to smash its own stack. For example, if a stray memory write (e.g., from an array overrun or pointer error) overwrites a return address on the call stack, then the program will return to a location unforeseen by the analysis. Our analysis assumes that return addresses are never overwritten. Our stack tool can be configured to emit a warning when it encounters an indirect store with indeterminate target, but, in practice, these warnings are not helpful to the user because there are so many of them.

3.5.2 *Self-Modifying Code.* The behavior of self-modifying code is very hard to statically analyze and, in general, even detecting such code reduces to the halting problem. Fortunately, the AVR has a Harvard architecture and all self-modifying code uses a special "store program memory" (`spm`) instruction. Our analyzer terminates with an error message if it finds an executable `spm` instruction; our rationale is that self-modifying code is likely to have stack behavior that cannot be statically analyzed. Fortunately, use of self-modifying code is rare and discouraged—it is notoriously difficult to understand and also precludes reducing the cost of an embedded system by putting the program into ROM.

3.5.3 *Modeling the Stack Pointer.* The `push` and `pop` instructions are easy to analyze: they atomically decrement and increment the stack pointer. However, on the AVR, explicit stack pointer modification is tricky because the 16-bit stack pointer is stored in a pair of 8-bit I/O registers. To modify the stack pointer atomically, a program must disable interrupts, change it, and then reenable interrupts. The problem for a static analyzer is in recognizing that the intermediate values that exist in between the two 8-bit writes are not "true" stack pointer values, in the sense that the intermediate address is never written to. Another problem is that it is difficult to distinguish between writes to the stack pointer that create a new stack (initializing the stack pointer), writes that are incrementing or decrementing the stack pointer, and writes that are switching to a new stack. The latter do not occur in the systems that we analyzed, but are found in embedded systems that run multiple threads. It would be relatively straightforward to perform a backward dataflow analysis on the values written into the stack pointer, to see where they come from, but our current forward-analyzing framework does not support this. The alternative approach that we have taken to solve both of these problems is to add several *macro instructions* to the instruction set recognized by our analysis tool. Each of these instructions is pattern-matched on a sequence of code generated by the compiler to increment, decrement, or initialize the stack pointer. Pattern-matching introduces a dependency on particular C compilers (and on the version used): if the compiler is changed, a few more macros may have to be added. This technique works in practice, because the set of idioms that the compiler uses to change the stack pointer is not large, and because C programmers usually have no reason to modify the stack pointer in a generic way. If the analyzer finds a store to the stack pointer that does not match any of its built-in patterns, it returns an unbounded stack depth. Finally, nonconstant changes to the stack pointer, for example those that implement the `alloca` call, can be handled as long as the value being added to or subtracted from the stack pointer can be identified by the static analysis.

3.5.4 *Indirect Branches.* Our analysis must build a conservative approximation of the program's control flow graph. Indirect branches cause problems for program analysis, because it can be difficult to tightly bound the set of potential branch targets. Our approach to dealing with indirect branches is based on the observation that they are usually used in a structured way and the structure can be exploited to learn the set of targets. For example, when analyzing TinyOS [Hill et al. 2000] programs, the argument to the function `TOS_post` is usually a literal constant representing the address of a function that will be called by an event-scheduling loop. The value of the argument is identified by abstract interpretation, but the connection between posting an event and making the indirect call must be established by adding (a small amount of) application-specific code to the analyzer. Automated analysis of this connection would be difficult, as it involves following data through a circular buffer in memory.

The stack analysis cannot deal with the form of indirect branch found in the context switch routine of a preemptive real-time operating system—the set of potential targets is too large. We do not address the problem of analyzing multithreaded programs in this paper, but we do not believe that it would be difficult. Since switching context to a new thread involves a change to a completely separate stack, writes to the stack pointer that restore a thread's state should simply be ignored, leading to separate analysis of threads. This approach is analogous to the way we currently analyze interrupt handlers separately.

3.5.5 *Recursion.* Engblom [1999] studied a collection of embedded systems containing over 300,000 lines of C code; it contained only 14 recursive loops. Therefore, our approach to dealing with recursion is blunt: we require that developers explicitly assert a maximum iteration count for each recursive loop in a system. These assertions, of course, must be externally verified, for example, by inspection or exhaustive testing of the recursive routine. Within the analysis, each recursive loop is unrolled to its maximum depth.

3.5.6 *Modeling Stack Frames.* Since the AVR architecture has plenty of general-purpose registers, there is little need to model values that are allocated in stack memory. In effect, we exploit the compiler's register allocation algorithm to implicitly specify the variables whose values should be modeled by our analyzer. Even so, a few AVR programs that we analyze could benefit from tracking values through stack-allocated memory and so we have added an experimental stack model to our analysis tool to model spilling of registers to the stack. This model is very simple and exploits the fact that our AVR compiler always produces matched `push` and `pop` instructions; we have never observed it to use an indirect memory reference to modify a value pushed onto the stack. The stack model is merged at control-flow merge points by merging individual stack elements; if two different-sized stacks are found at a merge point, the stack contents are invalidated. In the general case, this model is unsound: it relies on compiler conventions and so we require the developer to take special action to enable it.

3.6 Summary

We use an abstract interpreter to estimate the reachable states at each point in a program using a context-sensitive, path-insensitive, forward dataflow analysis of object code. Each bit is independently modeled using a three-valued logic: a bit can be 0, 1, or unknown. A bitwise analysis is used, because we are primarily interested in the interrupt enable/disable bits that are typically modified using bitwise operators. In order to produce results that are practically useful, we extended this sound base with some techniques that are empirically valid, but which are known, in general, to be unsound. These handle issues of indirect branches, recursion, spilling registers to the stack, and indirect stores.

4. STACK DEPTH ANALYSIS

The interrupt-preemption graph (IPG) is a weighted, directed graph, where each edge corresponds to a potential preemption by an interrupt handler and edge weights correspond to stack memory requirements. If a system permits, at most, one outstanding instance of each interrupt handler, the IPG will be acyclic and we can compute the maximum stack depth by finding the longest path through it. The property that permits this algorithm to work is that the stacking behavior of interrupt handlers is much simpler than, and can usually be treated independently of, the details of control flow and data flow within individual interrupt handlers.

4.1 Computing and Traversing the Interrupt Preemption Graph

Creating the interrupt preemption graph is straightforward once the dataflow analysis has successfully completed. The algorithm is as follows. For the main program and for each interrupt handler, examine all reachable instructions in order to compute:

- depth(i): The maximum contribution to the stack by interrupt handler i; this number is always nonnegative.
- depth(i, j): The maximum contribution to the stack by interrupt handler i at program points where interrupt handler j is enabled or negative infinity if interrupt handler j is never enabled by i.

This is a convenient representation of the IPG.

For an acyclic IPG with n interrupt vectors, the worst-case stack depth (wcsd) starting at any given interrupt handler can be computed as follows:

$$\text{wcsd}(i) = \max \begin{cases} \text{depth}(i) \\ \max_{j=1..n}(\text{depth}(i, j) + \text{wcsd}(j)) \end{cases} \tag{3}$$

To understand this equation, observe that the worst-case stack depth for a given interrupt handler may not involve any further preemptions; this is modeled by the top argument to the outside “max” function. On the other hand, the worst-case stack depth starting at some interrupt handler may be caused by a preemption; this is modeled by the bottom argument to the outside “max” function. The same logic can be applied recursively. Because AVR processors use vector zero for the the reset vector that leads to `main()`, and because the reset vector starts executing on an empty stack, the overall worst-case stack depth for a system is wcsd(0).

Our stack analysis algorithm has worst-case complexity exponential in the number of interrupt handlers. In principle, there may be embedded systems where interrupt analysis is quite expensive: some AVR processors support 34 interrupt vectors and some Texas Instruments DSPs in the TMS320 family support 96 separate interrupts. In practice, however, the analysis runs quickly:

all programs that we looked at declare nine or fewer interrupt handlers. For a model of interrupt-driven software similar to ours, Chatterjee et al. [2003] proved that bounding stack depth requires time exponential in the number of interrupt handlers, in the worst case.

4.2 Assumptions, Limitations, and Challenges

The stack depth analysis presented in this paper, like the dataflow analysis, is based on a number of assumptions. In this section we explain and justify these assumptions.

4.2.1 *Cyclic Interrupt Preemption Graphs.* Stack depth is always bounded for a system where the IPG is acyclic and individual interrupt handlers have bounded stack depth. On the other hand, the algorithm suggested by Eq. (3) does not terminate when there are cycles in the interrupt preemption graph. A cyclic IPG occurs, for example, when an interrupt handler runs with all interrupts enabled, admitting the possibility that a new instance of the interrupt will fire before the previous instance terminates. In this case, it superficially appears that an infinite chain of interrupt preemptions can occur, but this is not necessarily possible: each instance of the interrupt may terminate before the next is signaled. Proof of stack safety is much more difficult in the presence of cyclic preemptions: in effect, the stack bounding problem becomes predicated on the results of a much more difficult real-time analysis that is beyond the scope of this project.

Unfortunately, in embedded systems that we have examined, many interrupt handlers run with all interrupts enabled. Rather than returning an infinite stack bound for these programs, we provide developers with a way to turn a cyclic IPG into a DAG by manually asserting that a particular interrupt handler can preempt itself only up to a certain number of times. In practice, most systems that have cyclic IPGs should be considered to be poorly designed: few interrupt handlers are written in a reentrant fashion, so it is usually better to design systems where concurrent instances of a single handler are not permitted. Furthermore, stack depth requirements and the potential for race conditions will be kept to a minimum if there are no cycles in the interrupt preemption graph and if preemption of interrupt handlers is only permitted when necessary to meet real-time deadlines. We explore this idea further in Section 6.2.

There are a few situations where it is desirable to permit multiple outstanding instances of an interrupt handler. For example, the timer interrupt handler in the AvrX operating system [Barello 2004] was carefully designed to operate correctly when it preempts itself; this design makes it less likely that skew in the system clock will be introduced through missed timer interrupts.

4.2.2 *Effect of Interrupts on Machine State.* Our stack analysis assumes that the execution context is properly restored on return from interrupt. In other words, when an interrupt handler returns, the program counter, stack pointer, and general-purpose registers must contain the same values that they held just before the interrupt was signaled. For systems written in C, such as the ones we examine in Section 5, the machine state is automatically saved in the *interrupt prolog* and restored in the *interrupt epilog*—both are generated by the compiler, and developers need not make any effort to justify our assumption. For systems written in assembly, registers must be manually saved and restored. The most likely conditions under which this assumption is invalid, is for programs that use nonstandard extensions to C in order to allocate a global variable in a register or for interrupt handlers written in assembly language to exploit the provision of separate register banks for interrupt handlers (on processors like the Z80 and ARM).

Interrupts may modify machine state, such as memory and I/O registers that are outside of the processor's execution context. Our analyzer does not model main memory, but it does need to account for interrupts that alter the interrupt mask. For example, some interrupt handlers in TinyOS kernels enable the ADC (analog-to-digital converter) completion interrupt. We deal with this by detecting handlers that enable interrupts and then treating those interrupts as if their individual enable bits were permanently set. This could slightly overestimate the actual worst-case stack depth.

4.3 Using the Stack Tool

We have a prototype tool that implements our stack depth analysis. In its simplest mode of usage, the stack tool returns a single number: an upper bound on the stack depth for a system. For example:

```
$ ./stacktool -w flybywire.elf
total stack requirement from global analysis = 55
```

To make the tool more useful, we provide a number of extra features, including switching between context-sensitive and context-insensitive program analysis, creating a graphical callgraph for a system, listing branches that can be proved to be dead in all contexts, finding the shortest path through a program that reaches the maximum stack depth, and printing a disassembled version of the embedded program with annotations indicating interrupt status and worst-case stack depth at each instruction. These are all useful in helping developers understand and manually reduce stack memory consumption in their programs.

5. VALIDATION AND EVALUATION

We used several approaches to increase our confidence in the validity of our analysis techniques and their implementation.

5.1 Validating the Abstract Interpreter

To test the abstract interpreter, we modified the Atemu [2004] simulator for AVR processors to dump the state of the machine after executing each instruction. We then created a separate program to ensure that this concrete state was "within" the conservative approximation of the machine state produced by abstract interpretation at that address. We also ensured that the simulator did not execute any instructions that had been marked as dead code by the analyzer.

During early development of the analysis, this was helpful in finding bugs and in providing a much more thorough check on the abstract interpretation than manual inspection of analysis results—our next-best validation technique. We have tested the current version of the stack analysis tool by executing millions of instructions from a number of programs, including several that were written specifically to stress-test the analysis, without finding any discrepancies.

5.2 Validating Stack Bounds

There are two important metrics for validating the bounds returned by the stack tool. The first is qualitative: Does the tool ever return an unsafe result? Our second metric is quantitative: Is the tool capable of returning results that are close to the true worst-case stack depth for a system? The maximum observed stack depth, the worst-case stack depth estimate from the stack tool, and the (noncomputable) true worst-case stack depth are related in this way:

$$\text{worst-observed} \leq \text{true worst} \leq \text{estimated worst}$$

One might hope that the precision and accuracy of the analysis could be validated straightforwardly by running embedded codes in a simulator, logging their worst-observed stack depths, and comparing these values to the static bounds. For several reasons, this approach produces maximum observed stack depths that are significantly smaller than the estimated worst case and, we believe, the true worst case. First, the timing issues that we discussed in Section 1.1 come into play, making it very hard to observe interrupt handlers preempting each other even when it is clearly possible that they may do so. Second, even considering only sequential code, it can be very difficult to force an embedded system to execute the code path that produces the worst-case stack depth. Embedded systems often present a narrower external interface than do traditional applications, and it is correspondingly harder to force them to execute certain code paths using test inputs. While the difficulty of thorough testing is frustrating, it does support our thesis that static program analysis is particularly important in this domain. In other words, if it were easy to reach the worst-case stack depth during testing, static analysis would be far less important.

Since the global worst-case stack bound cannot be effectively validated empirically, we quantitatively evaluated the analysis of individual interrupts by instrumenting the AVR simulator to record the worst-case stack depth for each interrupt. For instance, consider the following example, using the TinyOS kernel CntToLedsAndRfm. This excerpt from the output of a tool that analyzes the output of the simulator shows the worst-observed stack depths for several interrupt handlers on one of the simulated nodes:

```
observed worst-case depths:
  vector  0 = 27
  vector 15 = 19
  vector 17 = 29
  vector 21 = 30
```

The corresponding output from the stack analysis tool is as follows:

```
vector  0 RESET = 29, at dfc
vector  1 INT0 = 2, at 4
vector 15 TIMER0_COMP = 19, at 1386
vector 17 SPI_STC = 29, at 1314
vector 18 USART0_RX = 21, at 1d38
vector 20 USART0_TX = 23, at 2082
vector 21 ADC = 30, at 1314
```

Comparing the empirical and analytical numbers, we can see that each interrupt that fired during the simulation run reached its worst-case stack depth, while the main context (vector zero) was two bytes short of its worst case. Two interrupts, 18 and 20, did not fire during the simulation and, in fact, the Atemu simulator does not yet support the serial port hardware that would cause these interrupts to be signaled. These results are typical: sequential pieces of code are often observed to closely approach or reach their worst-case depth, while the highly timing-dependent global worst case is seldom approached. We never observed an interrupt handler or an entire application to use more stack depth than its analytic bound; this justifies some confidence in the design and implementation of our analyzer.

5.3 Evaluating the Global Analysis

We evaluated the precision of our analysis tool using 75 embedded applications that fit into three categories. The first is a collection of application programs that are distributed with TinyOS version 0.6.1. TinyOS [Hill et al. 2000] is a small operating system for networked sensor nodes. Version 0.6.1 is obsolete; we include it for diversity—it was the last version written in C before the release of TinyOS 1.0, which was rewritten in nesC [Gay et al. 2003]. nesC is a programming language similar to C that is specifically designed to support component-based embedded software. The nesC language is compiled by translating into C. The second category of applications we analyze are those that ship with the most recent snapshot of TinyOS: 1.1.5. Finally, we have a few miscellaneous applications written for AVR microcontrollers: a control program for a self-balancing scooter and two versions of a simple control program for an autonomous helicopter developed by the Autopilot Project [2004]. All programs were compiled from C using gcc version 3.0.2 or 3.3.0, and all target various members of the Atmel AVR family of microcontrollers: the ATmega16, which has 1 KB of RAM and 16 KB of flash memory, and the ATmega103 and ATmega128, both of which have 4 KB of RAM and 128 KB of flash memory.

Of the 75 applications, there are four that defeat our analysis tool, all due to use of indirect jumps. The stack analysis results from the remaining 71 kernels are too large to display in a figure. Thus, we have chosen, at random, 15 TinyOS 0.6.1 applications and 15 TinyOS 1.1.5 applications and displayed them in Figure 6, along with the scooter and Autopilot results. Analysis of the results for all 71 applications shows that, on average, the whole program

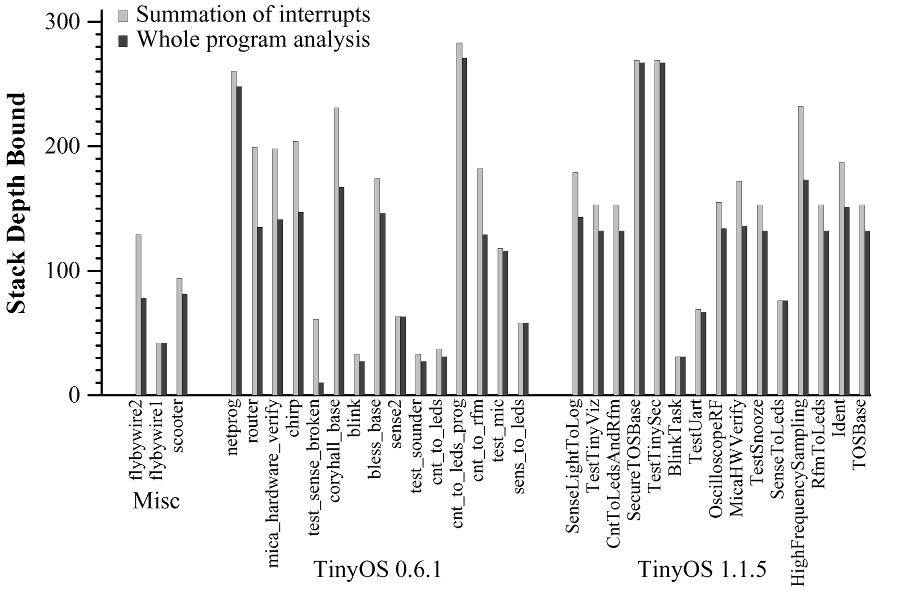


Fig. 6. Comparing stack bounds for summation of interrupts and whole-program analysis.

analysis returns a stack bound that is 19% lower than the bound returned by the naive summation of individual interrupt depths.

The whole program analysis failed to identify a concrete value for the master interrupt enable bit for only 2.2% of instructions, on average, across our test programs. The interrupt masks for nearly all of these instructions can be identified by turning on our experimental model of values stored in stack memory.

Since increased precision in the analysis translates directly into memory savings for embedded developers, we believe that the added complexity of the context-sensitive whole-program analysis is justified. In most cases where the more powerful analysis did not decrease the stack bound—for example, the "flybywire1" application—there was simply nothing that the tool could do. These applications run all interrupt handlers with interrupts enabled, precluding tight bounds on stack depth. Finally, only 13 of the 71 test kernels require more than 1 s to analyze on a 3.0 GHz Pentium 4, with the worst-observed run time of the analyzer on our inputs being a little under 9 s.

6. REDUCING STACK DEPTH

Previous sections described and evaluated our technique for bounding stack depth. In this section, we go a step further by exploring ways to reduce maximum stack memory requirements. Reducing stack depth is useful because it frees memory that can be used for other purposes, or potentially permits a product to be based on a cheaper CPU with less on-chip RAM.

Recall that the bound on stack depth is computed by finding the longest path through a DAG. There are two obvious ways to reduce the length of this path. First, the edge weights can be reduced without changing the shape of the graph. Second, the graph can be reshaped to contain fewer nodes and edges. We exploit

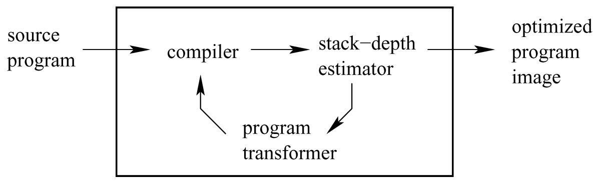


Fig. 7. Overview of stack depth reduction by inlining.

each of these strategies: edge weights can be reduced by generating code that makes more efficient use of stack memory and nodes can be removed from the interrupt preemption graph by eliminating unnecessary preemption relations. The first optimization is entirely automatic, while the second requires input from a developer who understands how the interrupt handlers in a particular system interact.

6.1 Driving Inlining Using Stack Bounds

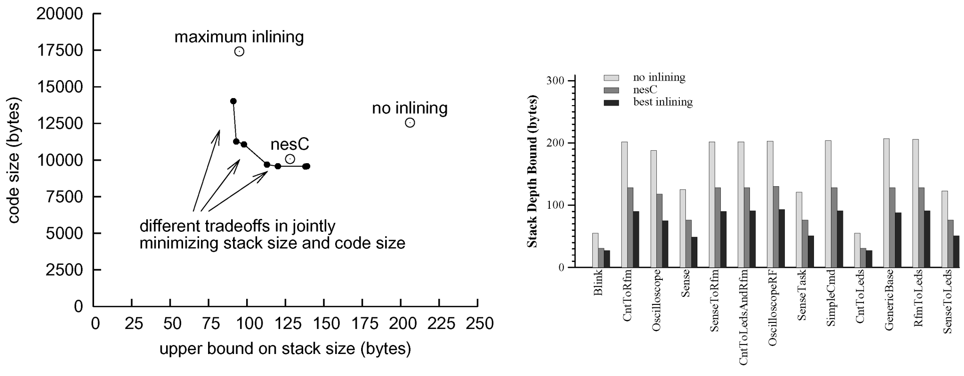
Given a way to quickly and accurately bound the stack depth of a program, it becomes possible for a compiler or similar tool to rapidly evaluate the effect of a large number of program transformations on the stack requirements of a system. We can then choose to apply only the transformations that improve stack memory usage.

Figure 7 illustrates our approach to automatic stack depth reduction. Although this technique is generic and would admit a variety of program transformations, so far the only transformation we have implemented is global function inlining. Inlining is a common optimization that replaces a function call with a copy of the function body. The immediate effect of function inlining on stack usage is to avoid the need to push a return address and function arguments onto the stack. More significantly, by cloning the body of a function, inlining permits the compiler to specialize the inlined code for its calling context. This may simplify code in such a way that fewer temporary variables are required, which may further reduce stack usage. Inlining also allows better register allocation since the compiler considers the caller and the callee together instead of separately.

In previous work [Reid et al. 2000], we developed a tool that performs inlining on C programs, even across multiple compilation units. To support the work reported here, we modified our inliner to accept an explicit list of callgraph edges to inline.

In general, inlining needs to be used sparingly. If a function is inlined many times, the size of the compiled binary can greatly increase. Furthermore, aggressive inlining can actually increase stack memory requirements by increasing the apparent live ranges of variables, causing the register allocator to spill variables onto the stack.

To evaluate a set of inlining decisions, we compute its *cost* using a user-specified function of stack depth and code size. For example, one obvious cost function minimizes stack depth without regard to code size. Another useful cost function is willing to trade one byte of stack memory for 32 bytes of code



(a) Different tradeoffs for RfmToLeds (b) Results for some TinyOS 1.1.5 kernels

Fig. 8. Comparing default compilation methods with stack reduction through inlining.

memory, since the processors we currently use have 32 times more code memory than data memory.

Systems that we have analyzed contain between 110 and 824 callgraph edges that could be inlined, leading in the worst case to 2^{824} possible inlining decisions. Since this space cannot be searched exhaustively, we use the following heuristic search:

1. Begin with an empty set of inlining decisions. Iterate through the list of callgraph edges, adding each edge to the set of inlinings only when this improves the cost metric. Repeat until the entire list can be scanned without adding any edges to the set.
2. Begin with a set of inlining decisions that contains all callgraph edges. Iterate through the list of callgraph edges, removing each edge from the set when this improves the cost metric. Repeat until the entire list can be scanned without removing any edges from the set.
3. Create a set of inlining decisions that contains every edge where the solutions from steps one and two agree and contains some of the edges where they disagree. Repeat until a sufficiently good solution is found or the time allocated to the search expires.

We arrived at this hybrid strategy after empirically observing that minimizing code size is often best accomplished by starting with no functions inlined, while minimizing stack depth is often best accomplished by starting with all functions inlined.

Figure 8a shows the results of applying the stack-depth/code-size reduction procedure to the TinyOS kernel RfmToLeds. There are three data points corresponding, respectively, to the system compiled without any function inlining, to the system compiled with as much inlining as possible (subject to limitations on inlining recursive functions and indirect calls), and to the system compiled by the nesC compiler [Gay et al. 2003], which performs fairly aggressive global function inlining on its own. It is interesting to note that while nesC's inlining heuristics are very good at reducing code size, neither of the extreme policies—maximal or zero inlining—results in a good tradeoff between stack usage and code size. The remaining data points were collected by running our stack reduction algorithm with a variety of cost functions, ranging from those that gave high priority to reducing stack depth to those that gave high priority to reducing code size. These results are typical: we applied stack depth reduction to a number of TinyOS kernels; the results are shown in Figure 8b. Our method reduces worst-case stack usage by up to 61% when compared to compilation without inlining and by up to 36% compared to kernels compiled using nesC.

The stack reduction program itself uses negligible CPU time, but it runs many trial compilations. The overall running time depends strongly on the size and structure of the program being transformed; for the example kernel used to generate Figure 8a, generating each data point took about 80 min. This means that unlike stack bounding, stack reduction through inlining is not presently usable in an interactive fashion, although it would be suitable for overnight runs.

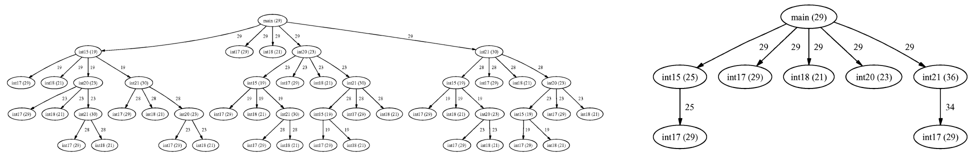
Our search heuristic could be sped up considerably by first applying standard inlining heuristics, such as inlining all functions with small bodies and all functions that are called only once. We opted not to do this because we have seen instances where these heuristics were wrong, in the sense that the cost function was made worse. Even so, we believe that it would be worth investigating faster heuristics that provide most of the benefit of the slow heuristic described in this section.

We found goal-driven inlining to be surprisingly effective at reducing stack memory usage. Why should this one optimization make such a large difference? First, inlining appears to serve as a "meta optimization": it facilitates, inhibits, and controls the scope of many other optimizations. Second, TinyOS is a component-based system that contains many more function calls than a functionally equivalent monolithic embedded system would. Systems with fewer function calls would not be as amenable to this technique, although more sophisticated partial-inlining schemes would be likely to work. Finally, the default compilation modes leave much room for improvement: gcc is a highly portable compiler that was not designed to support resource-constrained embedded processors. It would be interesting to investigate the question of whether it is possible to use advanced compilation techniques to generate code that makes very efficient use of stack memory without performing trial compilations.

6.2 Eliminating Unnecessary Preemption

In a well-designed embedded system, every edge in the interrupt preemption graph serves a specific purpose. Edges from the main context to interrupt handlers (i.e., edges like those in Figure 2a) must be present for interrupts to work at all, while edges from interrupt handlers to other interrupts are used to ensure that real-time deadlines are not missed. In other words, it might be necessary for an interrupt with tight response-time requirements to preempt a long-running interrupt handler.

Our experience is that many embedded developers create systems with too many interrupt preemption edges. In addition to using extra stack memory,



(a) Original (b) Restricted

Fig. 9. Interrupt preemption graphs for the RfmToLeds kernel from TinyOS 1.1.5.

	Original		Restricted		
Kernel name	Edges in IPG	Stack bound (bytes)	Edges in IPG	Stack bound (bytes)	Stack bound improvement
Blink	1	31	1	31	0%
RfmToLeds	47	128	7	92	28%
CntToLedsAndRfm	47	128	7	94	27%
HighFrequencySampling	79	161	11	127	21%
SecureTOSBase	129	267	7	220	18%

Fig. 10. Reducing stack depth by eliminating preemption relations.

unnecessary preemption incurs overhead in terms of cache pollution and added synchronization code and it also creates many more opportunities to implement race conditions.

In this section we propose and evaluate an empirical method for finding edges in the interrupt preemption graph that are candidates for removal. First, an embedded system is statically analyzed to compute a conservative estimate of its interrupt preemption graph. Second, the system is run in a simulator that has been instrumented to build a dynamic interrupt preemption graph based on actual preemptions that occur during simulation. Third, the interrupt preemption edges that are in the static IPG, but not in the dynamic IPG, are considered to be candidates for removal. An edge can be removed by modifying interrupt prolog and epilog code to mask off the appropriate interrupt enable bit. A developer who understands the details of interactions between interrupt handlers must carefully select the edges to be removed from the IPG, as this transformation can break a system if applied blindly.

Figure 9a shows the conservative IPG for the TinyOS kernel RfmToLeds. Figure 9b shows a much smaller IPG that corresponds both to the observed behavior of the unmodified RfmToLeds and also to the conservative IPG for the modified version that we created. Based on our understanding of RfmToLeds, omitting the IPG edges absent in the smaller IPG should lead to correct behavior. We tested the modified RfmToLeds kernel on a real TinyOS node and found that it works. The worst-case stack depths of the two kernels are 128 and 92 bytes, respectively—a 28% improvement. Furthermore, the stack bound for the original kernel was only valid under the assumption that interrupt handlers never preempt themselves; the stack bound for the modified system requires no such assumption. We applied this stack-reduction strategy to several other TinyOS kernels, ranging from very simple (Blink) to quite sophisticated (SecureTOSBase); the results of this study are shown in Figure 10.

```
... compiler-generated prolog ...

bool int15 = test_and_clear_15 ();      // record previous state of
bool int18 = test_and_clear_18 ();      // enable bits for interrupts
bool int20 = test_and_clear_20 ();      // 15, 18, 20, and 21, then
bool int21 = test_and_clear_21 ();      // disable them

__asm volatile ("sei");                 // set master interrupt bit

... body of interrupt handler ...

__asm volatile ("cli");                 // clear master interrupt bit

restore_15 (int15);                     // restore enable bits for
restore_18 (int18);                     // interrupts 15, 18, 20, 21
restore_20 (int20);                     // to their previous states
restore_21 (int21);

... compiler-generated epilog ...
```

Fig. 11. Code added to interrupt handlers 15 and 21 in the TinyOS RfmToLeds kernel to prune most of the edges leaving them in the interrupt preemption graph.

To implement the transformation that eliminates edges from the IPG, structured changes need to be made to the code implementing some interrupt handlers. Figure 11 shows the code that is added to interrupt vectors 15 and 21 in the RfmToLeds kernel to eliminate all outgoing IPG edges except those to vector 17. This code augments the compiler-generated prolog and epilog code: the prolog must clear some individual interrupt enable bits after saving their states, while the epilog must restore these bits to their previous states.

Though effective, our approach makes assumptions that may be unsafe, since it is possible that we might miss a rarely occurring combination of events that must be handled correctly in order to meet all of a system's implicit real-time deadlines. A more principled way to restrict interrupt preemption without violating real-time deadlines would be to use a real-time scheduling technique, such as preemption threshold analysis [Saksena and Wang 2000]. However, real-time analyses require inputs such as the deadline, minimum interarrival time, and worst-case execution time for each interrupt handler. This information is difficult to discover; we do not have it for the TinyOS software nor do we know of anyone who does.

7. RELATED WORK

Our dataflow analysis is largely a combination of standard techniques. Analysis of object code using a ternary logic appears to have first been done by Razdan and Smith [1994] and Java virtual machines perform an intraprocedural stack depth analysis [Lindholm and Yellin 1997]. Our contribution here lies in determining which combination of techniques obtains good results for our particular problem.

The previous research most closely related to our work is the stack depth analysis by Brylow et al. [2001]. Their analysis was designed to handle programs written by hand that are on the order of 1000 lines of assembly language; the programs we analyze, on the other hand, are compiled and are up to 30 times larger. Their contribution was to model interrupt-driven embedded systems, but their method could only handle immediate values loaded into the interrupt mask register—an ineffective technique when applied to software, such as TinyOS, where all data, including interrupt masks, moves through registers. Our work goes well beyond theirs through its use of an aggressive abstract interpretation of ALU operations, conditional branches, etc., to track the status of the interrupt mask. Also, Brylow et al. performed dataflow analysis and whole-program stack-depth analysis in a single pass, limiting the scalability of their approach. Our two-pass analysis is much more efficient, although it sacrifices some precision when analyzing systems where interrupt handlers enable other interrupts.

Palsberg and Ma [2002] provide a calculus for reasoning about interrupt-driven systems and a type system for checking stack boundedness. Like us, they provide a degree of context sensitivity (in their type system, this is encoded using intersection types). Unlike us, they model just the interrupt mask register, which would prevent them from accurately modeling our motivating example in Figure 3. The other major difference is that their focus is on the calculus and its formal properties and so they restrict their attention to small examples (10–15 instructions) that can be studied in detail. They also restrict themselves to a greatly simplified language that lacks pointers and function calls.

Chatterjee et al. [2003] analyze the time and space complexity of computing stack bounds for interrupt-driven programs. They consider a range of possible forms of interrupt handlers and show how the QSAT problem can be encoded in the interrupt preemption graph, yielding the dispiriting result that merely showing that a stack size bound exists is PSPACE-hard. This appears not to be an unlikely theoretical worst case: the complexity of our stack depth algorithm from Section 4 is exponential in the number of interrupt handlers, even for programs that do not encode QSAT problems in their interrupt structure.

AbsInt makes a commercial product called StackAnalyzer [Heckmann and Ferdinand 2002]. There is much overlap between our techniques and those underlying StackAnalyzer: they both rely on abstract interpretation of object code. At a higher level there is less overlap; StackAnalyzer supports multiple architectures, while our tool does not. It also includes a graphical interface for visualizing results, whereas our results are in text. On the other hand, the most important feature of our tool is that it supports integrated analysis of multiple interrupt handlers through estimation of the interrupt preemption graph; StackAnalyzer does not provide any explicit support for interrupt-driven systems.

Function inlining has traditionally been viewed as a performance optimization [Ayers et al. 1997] at the cost of a potentially large increase in code size. More recent work [Leupers and Marwedel 1999] has examined the use of inlining as a technique to reduce both code size and runtime. We are not aware of any previous work that uses function inlining specifically to reduce stack size or, in fact, of any previous work on automatically reducing the stack memory requirements of sequential code.

The relationship between the amount of preemption permitted by a system and its stack memory requirements has been known to exist for some time. For example, stack-based resource allocation [Baker 1990] is a protocol that permits logically concurrent activities to share a single stack, saving memory. More recently, Saksena and Wang's work on preemption threshold scheduling [2000] has permitted much more flexible tradeoffs between preemptive and nonpreemptive scheduling, providing fine-grained control over response time and stack memory usage. Finally, Gai et al. [2001] made use of preemption threshold scheduling to save memory on an embedded multiprocessor. All of these results have applied to thread-level scheduling; as far as we know there is no previous work on restricting interrupt preemption relations in order to reduce stack requirements.

8. CONCLUSION

The potential for stack overflow in embedded systems is hard to detect by testing. We have developed a practical static analysis that can show that an embedded system will not overflow its stack and demonstrated that the analysis provides accurate results. Experiments show that modeling the enabling and disabling of interrupt handlers using context-sensitive abstract interpretation produces estimates that are an average of 19% lower than estimates produced using the simpler approach of summing the individual stack requirements of interrupt handlers and the main function. We have also demonstrated two novel methods for reducing stack memory requirements. The first uses our analysis to drive a search for function inlining decisions that reduce stack depth. Experiments on a number of component-based embedded applications show that this approach reduces stack memory requirements by up to 36% when compared to aggressive global inlining without the aid of stack depth analysis. The second method is an empirical approach to eliminating unnecessary preemption relations from the interrupt preemption graph; we have shown that it reduces stack depth by up to 28%.

We feel that our experience has uncovered several interesting facts about embedded software for small processors. First, interrupt masks have a very static structure: for the vast majority of instructions, the interrupt mask has a known value across all possible executions, if the instruction's calling context is taken into account. Furthermore, a context-sensitive dataflow analysis based on a bitwise abstract interpretation of object code, is capable of efficiently discovering this static structure. Second, apparently innocuous features of an architecture or compiler can make analysis gratuitously difficult, as in the case of the nonatomic stack pointer manipulations discussed in Section 3.5, or gratuitously easy, as in the case of the Harvard architecture that makes self-modifying code into a non-issue. Finally, efficient extraction of useful results from static analysis of object code across a broad range of inputs is difficult. The design of the analyzer is more of an art than a science and many engineering compromises are required. In this paper, we have attempted to give a sense of the various tradeoffs we made and false paths we encountered, in order that subsequent developers of advanced tools for embedded systems can benefit from our experience.

Source code for the stack analyzer is available under `http://www.cs.utah.edu/~regehr/stacktool/`, and our global inliner can be downloaded from `http://www.cs.utah.edu/flux/knit/cmi.html`.

ACKNOWLEDGMENTS

The authors would like to thank Dennis Brylow, Eric Eide, Matthew Flatt, Wilson Hsieh, Jay Lepreau, and Michael Nahas for providing helpful feedback on drafts of this paper. This material is based upon work supported by the National Science Foundation under Grant No. 0209185.

REFERENCES

ALTER, D. 2003. Online stack overflow detection on the TMS320C28x DSP. Texas Instruments Application Note SPRA820. `http://www-s.ti.com/sc/psheets/spra820/spra820.pdf`.

ATEMU. 2004. Atemu: A sensor network emulator/simulator/debugger. Center for Satellite and Hybrid Communication Networks, University of Maryland. `http://www.cshcn.umd.edu/research/atemu/`.

AUTOPILOT PROJECT. 2004. `http://autopilot.sourceforge.net`.

AYERS, A., GOTTLIEB, R., AND SCHOOLER, R. 1997. Aggressive inlining. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, Las Vegas, NV. 134–145.

BAKER, T. P. 1990. A stack-based resource allocation policy for realtime processes. In *Proc. of the 11th IEEE Real-Time Systems Symp. (RTSS)*, Lake Buena Vista, FL. 191–200.

BALAKRISHNAN, G. AND REPS, T. 2004. Analyzing memory accesses in x86 executables. In *Proc. of the Intl. Conf. on Compiler Construction (CC)*, Barcelona, Spain.

BARELLO, L. 2004. The AvrX real time kernel. `http://barello.net/avrx`.

BRYLOW, D., DAMGAARD, N., AND PALSBERG, J. 2001. Static checking of interrupt-driven software. In *Proc. of the 23rd Intl. Conf. on Software Engineering (ICSE)*, Toronto, Canada. 47–56.

CHATTERJEE, K., MA, D., MAJUMDAR, R., ZHAO, T., HENZINGER, T. A., AND PALSBERG, J. 2003. Stack size analysis for interrupt-driven programs. In *Proc. of the 10th Static Analysis Symp.*, San Diego, CA. 109–126.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th Symp. on Principles of Programming Languages (POPL)*, Los Angeles, CA. 238–252.

DEBRAY, S., MUTH, R., AND WEIPPERT, M. 1998. Alias analysis of executable code. In *Proc. of the 25th Symp. on Principles of Programming Languages (POPL)*, San Diego, CA. 12–24.

ENGBLOM, J. 1999. Static properties of commercial embedded real-time programs, and their implication for worst-case execution time analysis. In *Proc. of the 5th IEEE Real-Time Technology and Applications Symp. (RTAS)*, Vancouver, Canada.

GAI, P., LIPARI, G., AND DI NATALE, M. 2001. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proc. of the 22nd IEEE Real-Time Systems Symp. (RTSS)*, London, UK.

GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. 2003. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, San Diego, CA. 1–11.

HECKMANN, R. AND FERDINAND, C. 2002. Stack usage analysis and software visualization for embedded processors. In *Proc. of the Embedded Intelligence Congress*, Nuremberg, Germany.

HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. 2000. System architecture directions for networked sensors. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA. 93–104.

LEUPERS, R. AND MARWEDEL, P. 1999. Function inlining under code size constraints for embedded processors. In *Proc. of the Intl. Conf. on Computer-Aided Design*, San Jose, CA. 253–256.

LINDHOLM, T. AND YELLIN, F. 1997. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA.

PALSBERG, J. AND MA, D. 2002. A typed interrupt calculus. In *Proc. of the 7th Intl. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, Oldenburg, Germany. 291–310.

RAZDAN, R. AND SMITH, M. D. 1994. A high-performance microarchitecture with hardware-programmable functional units. In *Proc. of the 27th Intl. Symp. on Microarchitecture (MICRO)*, San Jose, CA. 172–180.

REGEHR, J. AND REID, A. 2004. HOIST: A system for automatically deriving static analyzers for embedded systems. In *Proc. of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

REID, A., FLATT, M., STOLLER, L., LEPREAU, J., AND EIDE, E. 2000. Knit: Component composition for systems software. In *Proc. of the 4th Symp. on Operating Systems Design and Implementation*. Springer Verlag, San Diego, CA. 347–360.

SAKSENA, M. AND WANG, Y. 2000. Scalable real-time system design using preemption thresholds. In *Proc. of the 21st IEEE Real-Time Systems Symp. (RTSS)*, Orlando, FL.

Received May 2004; revised January 2005; accepted January 2005