In *Proceedings of the 3rd International Conference on Embedded Software*, Philadelphia, PA, pages 306–322, October 13–15, 2003.

Eliminating stack overflow by abstract interpretation

John Regehr Alastair Reid Kirk Webb

School of Computing, University of Utah

Abstract. An important correctness criterion for software running on embedded microcontrollers is *stack safety*: a guarantee that the call stack does not overflow. We address two aspects of the problem of creating stack-safe embedded software that also makes efficient use of memory: statically bounding worst-case stack depth, and automatically reducing stack memory requirements. Our first contribution is a method for statically guaranteeing stack safety by performing whole-program analysis, using an approach based on context-sensitive abstract interpretation of machine code. Abstract interpretation permits our analysis to accurately model when interrupts are enabled and disabled, which is essential for accurately bounding the stack depth of typical embedded systems. We have implemented a stack analysis tool that targets 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 a few seconds on the largest programs that we tested. The second contribution of this paper is a novel framework for automatically reducing stack memory requirements. We show that goal-directed global function inlining can be used to reduce the stack memory requirements of component-based embedded software, on average, to 40% of the requirement of a system compiled without inlining, and to 68% of the requirement of a system compiled with aggressive whole-program inlining that is not directed towards reducing stack usage.

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 the behavior of software running on these processors is important for two main reasons:

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

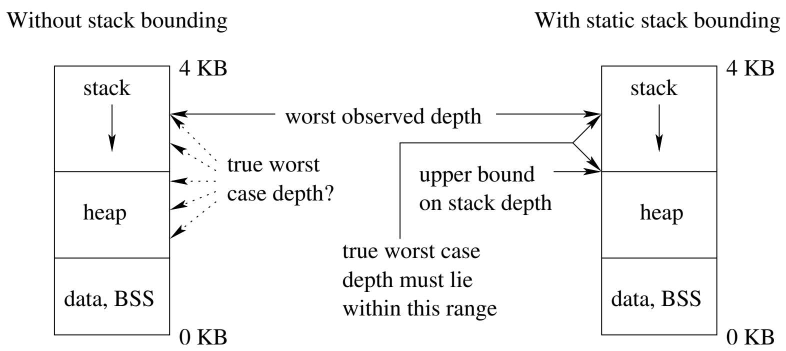


Fig. 1. Typical RAM layout for an embedded program with and without stack bounding. Without a bound, developers must rely on guesswork to determine the amount of storage to allocate to the stack.

In this paper we describe the results of an experiment in applying static analysis techniques to binary programs in order to bound and reduce their stack memory requirements. We check embedded programs for *stack safety*: the property that they 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 can easily crash a system. The transparent dynamic stack expansion that is performed by general-purpose operating systems is infeasible on small embedded systems due to lack of virtual memory hardware and limited availability of physical memory. 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 executable 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 alternative to static stack depth analysis that is currently used in industry is to ensure that memory allocated to the stack exceeds the largest stack size ever observed during testing by some safety margin. A large safety margin would provide good insurance against stack overflow, but for embedded processors used in products such as sensor network nodes and consumer electronics, the degree of overprovisioning must be kept small in order to minimize per-unit product cost. Figure 1 illustrates the relationship between the testing- and analysis-based approaches to allocating memory for the stack.

Testing-based approaches to software validation are inherently unreliable, and testing embedded software for maximum stack depth is particularly unreliable because its behavior is timing dependent: the worst observed stack depth depends on what code is executing when an interrupt is triggered and on whether further interrupts trigger 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 microseconds at maximum stack depth; 2) a timer interrupt fires 100 times a second spending 100 microseconds at maximum stack depth; and 3) a packet arrives on a network interface up to 10 times a second; the handler spends 100 microseconds at maximum stack depth. If these events occur independently of each other, then the worst case will occur roughly once every 10 years. This means that the worst case will probably not be discovered during testing, but will probably occur in the real world where there may be many instances of the embedded system. In practice, the events are not all independent and the timing of some events can be controlled by the test environment. However, we would expect a real system to spend less time at the worst-case stack depth and to involve more events.

Another 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. Static stack analysis, on the other hand, identifies the critical path through the system and also the maximum stack consumption of each function; this usually exposes obvious candidates for optimization.

Using our method for statically bounding stack depth as a starting point, we have developed a novel way to automatically reduce the stack memory requirement of an embedded system. The optimization proceeds by evaluating 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. Testing-based approaches to learning about system behavior, on the other hand, are slower and typically only explore a fraction of the possible state space.

Our work is preceded by a stack depth analysis by Brylow et al. [3] that also performs whole-program analysis of executable programs for embedded systems. However, 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 necessitated a more powerful approach based on context-sensitive abstract interpretation of machine code; we motivate and describe this approach in Section 2. Section 3 discusses the problems in experimentally validating the abstract interpretation and stack depth analysis, and presents evidence that the analysis provides accurate results. In Section 4 we describe the use of a stack bounding tool to support automatically reducing the stack memory consumption of an embedded system. Finally, we compare our research to previous efforts in Section 5 and conclude in Section 6.

2 Bounding Stack Depth

Embedded system designers typically try to statically allocate resources needed by the system. 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 the machine state onto the stack before calling a user-defined interrupt handler, and pops the machine state upon termination of the handler. For developers of embedded systems, it is important not only to know that the stack depth is bounded, but also to have a tight bound — one that is not much greater than the true worst-case stack depth. This section describes the whole-program analysis that we use to obtain tight bounds on stack depth.

Our prototype stack analysis tool targets programs for the Atmel AVR, a popular family of microcontrollers. We chose to analyze binary program images, 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, and a safe analysis must account for its effects.
- The source code for libraries and real-time operating systems are commonly not available for analysis.
- Since the 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 non-standard language extensions that embedded compilers commonly use to provide developers with fine-grained control over processor-specific features.
- Adding a post-compilation analysis step to the development process presents developers with a clean usage model.

2.1 Analysis Overview and Motivation

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 [4], 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.

The second, more difficult, challenge in embedded systems is accurately estimating interactions between interrupt handlers and the main program to compute a maximum stack depth for the whole system. If interrupts are disabled while running interrupt handlers, one can safely estimate the stack bound of a system containing n interrupt handlers using this formula:

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

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 concurrent instance of each interrupt handler, the worst-case stack depth of a system can be computed using this formula:

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

```
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. 2. 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

Unfortunately, as we show in Section 3, this simple formula often provides unnecessarily pessimistic answers when used to analyze real systems where only some parts of some interrupt handlers run with interrupts enabled.

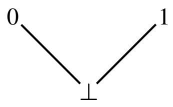
To obtain a safe, tight stack bound for realistic embedded systems, we developed a two-part analysis. The first must generate an accurate estimate of the state of the processor's interrupt mask at each point in the program, and also the effect of each instruction on the stack depth. The second part of the analysis — unlike the first — accounts for potential preemptions between interrupts handlers and can accurately bound the global stack requirement for a system.

Figure 2 presents a fragment of machine code that motivates our approach to program analysis. Analogous code can be found in almost any embedded system: 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. There are a number of challenges in analyzing such code.

First, effects of arithmetic and logical operations must be modeled with enough accuracy to track data movement through general-purpose and special-purpose registers. In addition, partially unknown data must be modeled. For example, analysis of the code fragment must 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 processor state at subsequent addresses.

Finally, to prevent procedural aliasing from degrading the estimate of the machine state, a context sensitive analysis must be used. For example, in some systems the code



(a) Lattice for each bit in the machine state

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	⊥
⊥	⊥	⊥	⊥

merge

	1	0	⊥
1	1	⊥	⊥
0	⊥	0	⊥
⊥	⊥	⊥	⊥

(b) Logical operations on abstract bits and combining machine states at merge points

Fig. 3. Modeling machine states and operations in the abstract interpretation

in Figure 2 is called with interrupts disabled by some parts of the system and is called with interrupts enabled by other parts of the system. With a context-insensitive approach, the analysis concludes that since the initial state of the interrupt flag can vary, the final state of the interrupt flag can also vary and so analysis of both callers of the function would proceed with the interrupt flag unknown. This can lead to large overestimates in stack bounds since unknown values are propagated to any code that could execute after the call. With a context-sensitive analysis the two calls are analyzed separately, resulting in an accurate estimate of the interrupt state.

The next section describes the abstract interpretation we have developed to meet these challenges.

2.2 Abstracting the Processor State

The purpose of our abstract interpretation is to generate a safe, precise estimate of the state of the processor at each point in the program; this is a requirement for finding a tight bound on stack depth. Designing the abstract interpretation boils down to two main design decisions.

First, how much of the machine state should the analysis model? For programs that we have analyzed, it is sufficient to model the program counter, general-purpose registers, and several I/O registers. Atmel AVR chips contain 32 general-purpose registers and 64 I/O registers; each register stores eight bits. From the I/O space we model the registers that contain interrupt masks and the processor status register. We do not model main memory or most I/O registers, such as those that implement timers, analog-to-digital conversion, and serial communication.

Second, what is the abstract model for each element of machine state? We chose to model the machine 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 and this turned out to lose too much information through conservative approximation. Each bit of machine state is modeled using the lattice depicted in Figure 3(a). The lattice contains the values 0 and 1 as well as a bottom element, $\perp$, that corresponds to a bit that cannot be proven to have value 0 or 1 at a particular program point.

Figure 3(b) shows abstractions of some common logical operators. Abstractions of operators should always return a result that is as accurate 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 executing 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 2 illustrates two 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. 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.

2.3 Managing Abstract Processor States

An important decision in designing the analysis was when to create a copy of the abstract machine state at a particular program point, as opposed to merging two abstract states. The merge operator, shown in Figure 3(b), is lossy since a conservative approximation must always be made. We have chosen to implement a context-sensitive analysis, which means that we fork the machine state each time a function call is made, and at no other points in the program. This has several consequences. First, and most important, it means that the abstract interpretation is not forced to make a conservative approximation when a function is called from different points in the program where the processor is in different states. In particular, when a function is called both with interrupts enabled and disabled, the analysis is not forced to conclude that the status of the interrupt bit is unknown inside the function and upon return from it. Second, it means that we cannot show termination of a loop implemented within a function. This is not a problem at present since loops are irrelevant to the stack depth analysis as long as there is no net change in stack depth across the loop. However, it will become a problem if we decide to push our analysis forward to bound heap allocation or execution time. Third, it means that we can, in principle, detect termination of recursion. However, our current implementation rarely does so in practice because most recursion is bounded by values that are stored on the stack — which our analysis does not model. Finally, forking the state at function calls means that the state space of the stack analyzer might become large. This has not been a problem in practice; the largest programs that we have analyzed cause the analyzer to allocate about 140 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.

2.4 Abstract Interpretation and Stack Analysis Algorithms

The program analysis 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 address of a startup routine that eventually jumps to `main()`. For each item in the worklist, the analyzer abstractly interprets a single instruction. If the interpretation 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 state space for a program is finite and because we never revisit states more than once.

The abstract interpretation detects control-flow edges that are dead in a particular context, and also control-flow edges that are dead in all contexts. In many systems we have analyzed, the abstract interpretation finds up to a dozen branches that are provably not taken. This illustrates the increased precision of our analysis relative to the dataflow analysis that an optimizing compiler has previously performed on the embedded program as part of a dead code elimination pass.

In the second phase, the analysis considers there to be a control flow edge from every instruction in the program to the first instruction of every interrupt handler that cannot be proven to be disabled at that program point. An interrupt is disabled if either the master interrupt bit is zero or the enable bit for the particular interrupt is zero. Once these edges are known, the worst-case stack depth for a program can be found using the method developed by Brylow et al. [3]: perform a depth-first search over control flow edges, explicit and implicit, keeping track of the effect of each instruction on the stack depth, and also keeping track of the largest stack depth seen so far.

A complication that we have encountered in many real programs is that interrupt handlers commonly run with all interrupts enabled, admitting the possibility that a new instance of an interrupt handler will be signaled before the previous instance terminates. From an analysis viewpoint reentrant interrupt handlers are a serious problem: systems containing them cannot be proven to be stack-safe without also reasoning about time. In effect, the stack bounding problem becomes predicated on the results of a real-time analysis that is well beyond the current capabilities of our tool.

In real systems that we have looked at reentrant interrupt handlers are so common that we have provided a facility for working around the problem by permitting a developer to manually assert that a particular interrupt handler can preempt itself only up to a certain number of times. Programmers appear to commonly rely on ad hoc real-time reasoning, e.g., "this interrupt only arrives 10 times per second and so it cannot possibly interrupt itself." In practice, most instances of this kind of reasoning should be considered to be design flaws — 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 a real-time deadline.

2.5 Other Challenges

In this section we address other challenges faced by the stack analysis tool: loads into the stack pointer, self-modifying code, indirect branches, indirect stores, and recursive function calls. These features can complicate or defeat static analysis. However, embedded developers tend to make very limited use of them, and in our experience static analysis of real programs is still possible and, moreover, effective.

We support code that increments or decrements the stack pointer by constants, for example to allocate or deallocate function-scoped data structures. Code that adds non-constants to the stack pointer (e.g., to allocate variable sized arrays on the stack) would require some extra work to bound the amount of space added to the stack. We also do not support code that changes the stack pointer to new values in a more general way, as is done in the context switch routine of a preemptive operating system.

The AVR has a Harvard architecture, making it possible to prove the absence of self-modifying code simply by ensuring that a program cannot reach a "store program memory" instruction. However, by reduction to the halting problem, self-modifying code cannot be reliably detected in the general case. 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.

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 [6] 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 manually. Making it easier to express such connections is an area of future work.

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. However, these branches need not be analyzed: since switching context to a new thread involves a change to a completely separate stack, it suffices to learn the worst-case stack usage of the operating system code and add it to the worst-case stack usage for each thread.

Memory writes can compromise our static analysis in two ways. First, an out-of-bounds store may overwrite a return address on the call stack, causing the program to return to an unforeseen location. Second, since the AVR maps its registers into low memory, a stray write could change a register in a way invisible to the analysis. We deal with direct stores by ensuring that they reference an appropriate range of RAM that is not occupied by the registers or call stack. Indirect stores are simply assumed not to overwrite a register or return address; our rationale is that a program containing this behavior is so flawed that its stack safety is irrelevant. In the long run a better solution would be to construct embedded software in a type-safe language.

Recursive code is uncommon in embedded software. For example, Engblom [4] studied a collection of embedded systems containing over 300,000 lines of C code, and it contained only 14 recursive loops. Our approach to dealing with recursion, therefore, is blunt: we require that developers explicitly specify a maximum iteration count for each recursive loop in a system. The analysis returns an unbounded stack depth if the developers neglect to specify a limit for a particular loop.

It would be straightforward to port our stack analyzer to other processors: the analysis algorithms, such as the whole-program analysis for worst-case stack depth, operate on an abstract representation of the program that is not processor dependent. However, the analysis would return pessimistic results for register-poor architectures such as the Motorola 68HC11, since code for those processors makes significant use of the stack, and stack values are not currently modeled by our tool. In particular, we would probably not obtain precise results for code equivalent to the code in Figure 2 that we used to motivate our approach. To handle register-poor architectures we are developing an approach to modeling the stack that is based on a simple type system for registers that are used as pointers into stack frames.

2.6 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 proven 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.

There are other obvious ways to use the stack tool that we have not yet implemented. For example, using stack bounds to compute the maximum size of the heap for a system so that it stops just short of compromising stack safety, or computing a minimum safe stack size for individual threads in a multi-threaded embedded system. Ideally, the analysis would become part of the build process and values from the analysis would be used directly in the code being generated.

3 Validating the Analysis

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

3.1 Validating the Abstract Interpretation

To test the abstract interpretation, we modified a simulator for AVR processors to dump the state of the machine after executing each instruction. Then, we 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, and that the simulator did not execute any instructions that had been marked as dead code by the static analysis. 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 at least 100,000 instructions of about a dozen programs, including several that were written specifically to stress-test the analysis, and did not find any discrepancies.

3.2 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? Testing the stack tool against actual execution of about a dozen embedded applications has not turned up any examples where it has returned a bound that is less than an observed stack depth. This justifies some confidence that our algorithms are sound.

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 (non-computable) 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 of the analysis could be validated straightforwardly by instrumenting some embedded systems to make them report their worst observed stack depth and comparing these values to the bounds on stack depth. 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 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 within the main function and individual interrupt handlers, 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.

The 71 embedded applications that we used to test our analysis come from three families. The first is Autopilot, a simple cyclic-executive style control program for an autonomous helicopter [10]. The second is a collection of application programs that are distributed with TinyOS version 0.6.1, a small operating system for networked sensor nodes. The third is a collection of application programs that are distributed with TinyOS 1.0 [6]. Version 1.0 is a complete rewrite of TinyOS using nesC [5], a programming language very similar to C that is compiled by translating into C. All programs were compiled from C using gcc version 3.0.2 or 3.1.1, and all target the ATmega103 chip, a member of the Atmel AVR family that contains 4 KB of RAM and 128 KB of flash memory.

3.3 Validating Analysis of Individual Interrupts

To quantitatively evaluate the stack tool, we wrote a program that modifies the assembly language version of an AVR program in such a way that each interrupt is handled on its own stack. This makes stack measurement timing-independent, but still leaves the difficult problem of making the main function and each interrupt handler execute the path to the worst-case stack depth.

We found that a perfect match between predicted and actual stack depth could only be obtained for slightly modified versions of simple embedded applications such as the `BlinkTask` TinyOS kernel whose function is to flash an LED. Even for this example, we were forced to comment out a call to a function supporting one-shot timers in a timer module: it contributed to the worst-case stack depth, but could never be called in our system. After making this small modification and adding serial-line driver code to enable reporting of stack depths to a separate computer, the `BlinkTask` application contained about 4000 lines of C code, even after a dead-code elimination pass performed by nesC. Running the stack analysis on this modified kernel produced the following results:

```
Stack depths:
   vector  0 main                = 33, at d30
   vector 15 _output_compare0_   = 32, at 50a
   vector 18 _uart_recv_         = 27, at 3e8
   vector 20 _uart_trans_        = 23, at a90
```

This shows the estimated worst-case stack depth of each entry point into the program and also an address at which this depth is reached. We then ran this kernel on an AVR processor and queried it to learn the worst observed stack depth; it reported the same stack depths as the analysis reported.

3.4 Evaluating the Global Analysis

Of the 71 applications used to test our analysis, there are seven that defeat our analysis tool, for example because they make an indirect jump based on a value that is not a literal constant, or they load an indeterminate value into the stack pointer. We believe that some of these applications could be successfully analyzed by a slightly improved version of our stack tool, but for now we disregard them.

The stack analysis results from the remaining 64 kernels are too large to display in a figure so we have chosen, at random, 15 TinyOS 0.6.1 applications and 15 TinyOS 1.0 applications and displayed them in Figure 4. Analysis of the results for all 64 applications shows that on average, the context insensitive global analysis returns a stack bound that is 15% lower than the bound determined by summing the requirements of

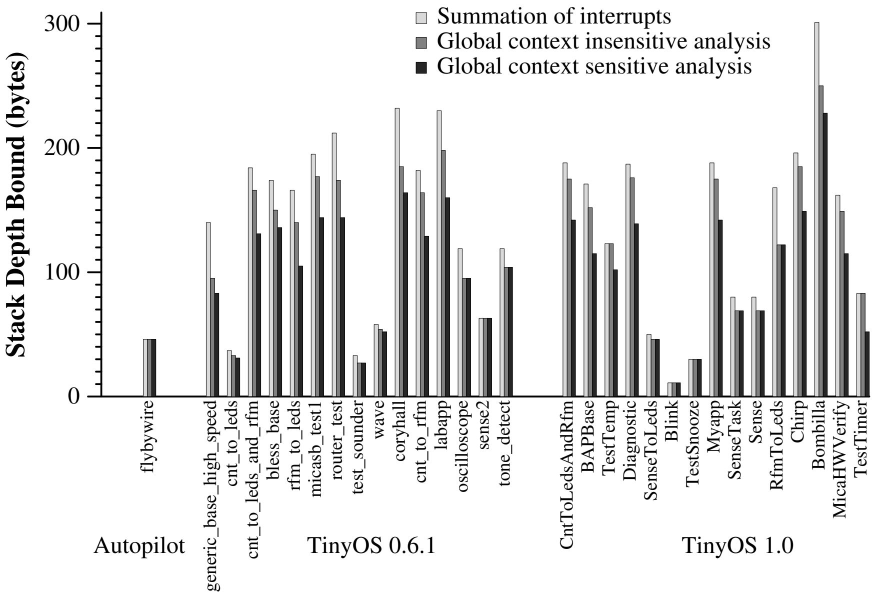


Fig. 4. Comparing stack bounds for summation of interrupts, global context insensitive analysis, and context sensitive analysis

interrupt handlers and the main function, and that on average the context sensitive analysis returns a bound that is 35% lower than the bound computed by summing interrupts.

Another metric that can be used to compare the context sensitive and insensitive analyses is the number of instructions for which each method fails to identify a concrete interrupt mask. The context insensitive analysis was unable to identify a concrete interrupt mask for 41% of the instructions in a given kernel, on average. The context sensitive analysis, on the other hand, failed to identify a concrete value for the interrupt mask for only 2.2% of instructions, which mostly fall into two categories. First, instructions whose interrupt mask would become analyzable if the call stack were modeled. Second, instructions whose interrupt mask is genuinely ambiguous — within a single context they can be shown to execute both with interrupts disabled and enabled.

Since increased precision in the analysis translates directly into memory savings for embedded developers, we believe that the added complexity of the context-sensitive analysis is justified. In most cases where the more powerful analysis did not decrease the stack bound — for example, the Autopilot 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, the stack depth analysis requires less than four seconds of CPU time on a 1.4 GHz Athlon for all of our example programs, and for many applications it requires well under one second.

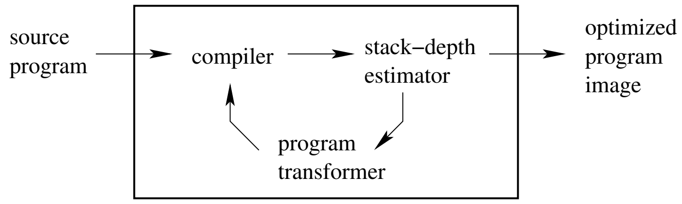


Fig. 5. Overview of stack depth reduction

4 Reducing Stack Depth

Previous sections described and evaluated a tool for bounding stack depth. In this section we go a step further by exploring the use of the stack bounding tool as an essential part of a method for automatically reducing the stack memory requirements of embedded software. Reducing stack depth is useful because it potentially frees up more storage for the heap, permits more threads to be run on a given processor, or permits a product to be based on a less expensive CPU with fewer on-chip resources.

The basic observation that makes stack minimization possible is that 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 then choose to apply only the transformations that improve stack memory usage.

Figure 5 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 experience with 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, inlining allows intraprocedural optimizations to apply which may simplify the code to the extent 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 simultaneously instead of separately. In general, inlining needs to be used sparingly. If a function is inlined many times, the size of the compiled binary can increase. Furthermore, aggressive inlining can actually increase stack memory requirements by overloading the compiler's register allocator. In previous work [11], we developed a global function inliner for C programs that can perform inlining on complete programs instead of within individual C files. To support the work reported in this paper, we modified this inliner to accept an explicit list of callgraph edges to inline.

To reduce the stack depth requirements of an embedded system we perform a heuristic search that attempts to minimize the *cost* of a program, where cost is a user-supplied 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 memory, since the processors we currently use have 32 times more code memory than data memory.

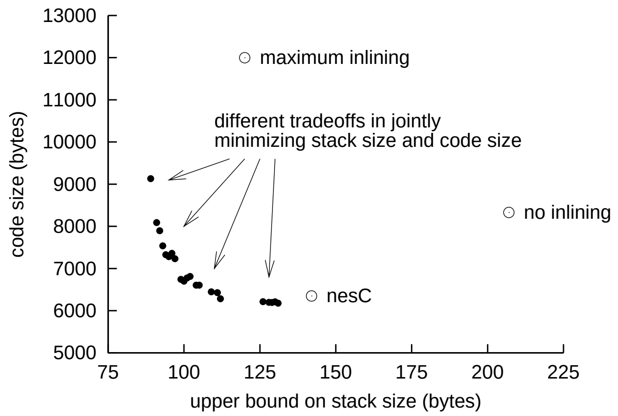


Fig. 6. Comparing stack reduction tradeoffs with default compilation methods for an example TinyOS kernel. Note that the origin of the graph is not at 0, 0.

Systems that we have analyzed contain between 80 and 670 callgraph edges that could be inlined, leading in the worst case to 2^{670} possible inlining decisions. Since this space obviously cannot be searched exhaustively, we use a heuristic search. We have found that an effective approach is to bound the degree of inlining "from above" and "from below" and then perform a random search of the space in between. Minimizing code size is often best accomplished by starting with no functions inlined and then repeatedly picking an uninlined function and inlining it only if this improves the cost metric. Minimizing stack depth, on the other hand, is often best accomplished by starting with all functions inlined and then repeatedly picking an inlined function and dropping the inlining if this improves the cost metric. To see why this is more effective at reducing stack depth, consider a system where there are two paths to the maximum stack depth. Separately considering inlining decisions on either path will not improve the stack depth: it is only by considering both inlinings at once that their benefit is seen.

Having found upper and lower bounds on the inlining decisions, we search the space between the bounds by accepting inlinings where the previous solutions agreed, and then repeatedly test inlinings that they disagreed on. In practice this step often finds solutions missed by the previous two steps.

Figure 6 shows the results of applying the stack depth / code size reduction algorithm to the TinyOS kernel `CntToLedsAndRfm`. There are three data points corresponding respectively to a system compiled without any function inlining, to a system compiled with as much inlining as possible (subject to limitations on inlining recursive functions and indirect calls), and to a system compiled by the nesC compiler [5], which performs fairly aggressive global function inlining on its own. 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 and found that we could usually use about 40% of the stack required by a kernel without any inlining, and about 68% of the stack required by kernels compiled using nesC.

5 Related Work

The previous research most closely related to our work is the stack depth analysis by Brylow et al. [3]. 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 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.

Palsberg and Ma [9] 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 2. 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, and they restrict themselves to a greatly simplified language that lacks pointers and function calls.

AbsInt makes a commercial product called StackAnalyzer [1]; its goal is to estimate stack depth in embedded software. We were not able to find much information about this tool. In particular, there is no indication that it is attempting to model the status of the interrupt mask, the most important feature of our analysis.

Our abstract interpretation is largely a combination of standard techniques. Java virtual machines perform an intraprocedural stack depth analysis [8], and program modeling at the bit-level has been done before (see, for example, the discussion of possible lattices for MIT's Bitwise project [12]). Our contribution here lies in determining which combination of techniques obtains good results for our particular problem.

Function inlining has traditionally been viewed as a performance optimization [2] at the cost of a potentially large increase in code size. More recent work [7] 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 embedded software.

6 Conclusion

The potential for stack overflow in embedded systems is hard to detect by testing. We have developed a static analysis that can prove 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 35% lower than estimates produced using the simpler approach of summing the stack requirements of interrupt handlers and the main function. We have also demonstrated a novel use of this 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 an average of 32% compared with aggressive global inlining without the aid of a stack depth analysis.

Availability: Source code for the stack analyzer is available under `http://www.cs.utah.edu/flux/stacktool`, and the global inliner can be downloaded from `http://www.cs.utah.edu/flux/alchemy/software.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.

References

1. AbsInt. StackAnalyzer. `http://www.absint.com/stackanalyzer`.
2. Andrew Ayers, Robert Gottlieb, and Richard Schooler. Aggressive inlining. In *Proc. of Programming Language Design and Implementation*, pages 134–145, Las Vegas, NV, June 1997.
3. Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt-driven software. In *Proc. of the 23rd Intl. Conf. on Software Engineering*, pages 47–56, Toronto, Canada, May 2001.
4. Jakob Engblom. 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.*, Vancouver, Canada, June 1999.
5. David Gay, Phil Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of Programming Language Design and Implementation*, pages 1–11, San Diego, CA, June 2003.
6. Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Cambridge, MA, November 2000.
7. Rainer Leupers and Peter Marwedel. Function inlining under code size constraints for embedded processors. In *Proc. of the International Conference on Computer-Aided Design*, pages 253–256, San Jose, CA, November 1999.
8. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, January 1997.
9. Jens Palsberg and Di Ma. A typed interrupt calculus. In *Proc. of Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 291–310. Springer Verlag, 2002.
10. The Autopilot Project. `http://autopilot.sourceforge.net`.
11. Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proc. of the 4th Symp. on Operating Systems Design and Implementation*, pages 347–360, San Diego, CA, October 2000.
12. Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proc. of Programming Language Design and Implementation*, pages 108–120, Vancouver, Canada, June 2000.