Eliminating Stack Overflow by Abstract Interpretation

Paper by John Regehr, Alastair Reid, and Kirk Webb
ACM Transactions on Embedded Computing Systems, Nov 2005

Truong Nghiem
nghiem@seas.upenn.edu

1

Outline

- Introduction
- Static Analysis Approach
- Dataflow Analysis
- Stack Depth Analysis
- Evaluation
- Reducing Stack Depth
- Conclusion

2

Introduction: Static Analysis

- Embedded software on inexpensive microprocessors:
  - Used in safety critical applications, hard to upgrade
  - Undesirable to overprovision resources due to strict constraints on cost, size, power
  - ➲ **Static analysis of software is important!**
  - ➲ **Worst-case resource requirements (e.g. memory) should be determined *statically* & accurately!**
- Why *statically* (but not *dynamically*)? Lack of resources and power on inexpensive microprocessors; real-time requirements

3

Introduction: Stack Safety

- *Stack safety* = not run out of stack memory at runtime
- Why is *stack safety* important? Stack overflow causes memory corruption

system crash / incorrect behavior
+
safety critical applications
= disaster

- Dynamic stack checking/expansion/sharing (see Wei's presentation) is infeasible on *small real-time* embedded systems.

4

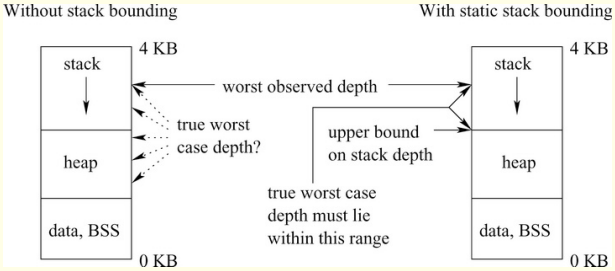
Introduction: Testing-based Approach

- Offline testing using as many cases as possible
- Typical procedure:
  - Set sufficiently large stack size
  - Run many tests. If stack overflow occurs, increase stack size.
  - Finally, use *larger* stack size for safety (hopefully)
- Disadvantages:
  - Some executable paths are missed (esp. in interrupt-driven systems) → *possibility of stack overflow not eliminated*
  - Programmers usually use larger stack size than needed → *waste of limited, valuable memory resources*
  - *Little feedback to help optimize memory usage*

5

Introduction: Static Analysis Approach

- Statically analyze the software to find stack bound
  - Never underestimate the true worst-case stack depth → ensure stack safety
  - Sufficiently precise: try to be as close to the true worst-case stack depth as possible, but …
  - Fast enough
  - Informative: to help improve memory usage
- Current: experimental tool for Atmel AVR microcontrollers.



6

Static Analysis Approach

- Stack requirement analysis:
  - Sequential code: rather straightforward
  - Interrupt-driven code: challenging
- An interrupt can fire at virtually any time, when:
  - The event (interrupt source) occurs
  - Typically: the *master interrupt enable bit* is set
  - The interrupt is enabled
  - Processor has just finished executing an instruction
- Finding stack depth bound: with *n* interrupts
  - Only one interrupt at a time → underestimate, not practical

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

  - All interrupts at a time → overestimate since not all interrupts are enabled at a time (consider interrupt masks)

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

7

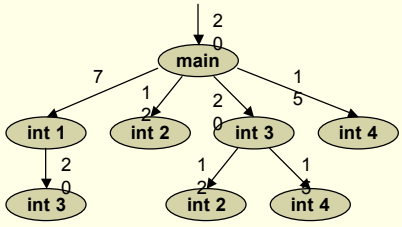
Static Analysis Approach

Two-part analysis:

- Context-sensitive dataflow analysis: identify unexecutable branches, compute state of interrupt mask at each point, compute worst-case stack depth of sequential code (main code, interrupt handlers)
- Compute *interrupt preemption graph (IPG)*

IPG: weighted, directed graph of possible interrupt preemptions

- ❖ Edges: potential preemptions
- ❖ Weights: stack memory requirements



➔ compute stack depth bound by searching through the IPG

8

Static Analysis Approach

- Analyze object code instead of source code:
  - Separated from compilers: we can use any compiler
  - Independent of languages (Asm, C, Pascal, ADA,…)
  - Don't need source code of libraries

9

Dataflow Analysis

- Two challenges:
  - Compute the stack requirement of each interrupt handler and the main code: call-graph can be constructed because indirect calls and recursion are uncommon in embedded systems [Engblom]
  - Compute the interrupt masks (master and individual) at each program point → to construct the IPG
- Tracking the interrupt masks:
  - Model the effects of arithmetic and logical operations
  - Model partially unknown data
  - Follow the control-flow: conditional instructions

  ➲ The more precise the interrupt mask tracking is, the better the computed stack bound is

10

Dataflow Analysis

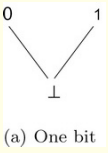
- Main idea: Abstracting the machine state
  - Abstract necessary part of the machine state (registers, memory,…)
  - Interpret the object code to approximate the machine state with *sufficient precision* in *reasonable time*
- Abstract interpreter – design questions:
  - What should be modeled? *Program counter, general-purpose registers, several SFRs, probably values on the stack (experimental). Main memory and most SFRs are not modeled.*
  - How to model? *Model at the bit-level to capture bitwise operations on interrupt masks and condition code register.*

11

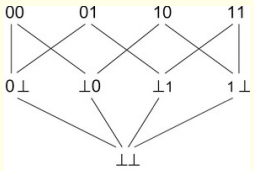
Dataflow Analysis

Bit-level modeling of machine state

- Model each bit by set of three values
  - 0: bit is 0
  - 1: bit is 1
  - ⊥: unknown value
- Unknown values
  - External input, e.g. sensor reading
  - Data from unmodeled part, e.g. main memory
  - Mergence of control flow paths, e.g. if-then-else



(a) One bit



(b) Two bits

12

Dataflow Analysis

Operations in the bit-wise domain

- Results of abstract operators should be as precise as possible, considering the unknown value of bits.
- Two special cases:
  - Add-with-carry instruction: consider carry bit, if applied to same register then it is “rotate left through carry”

```
adc   r24, r24 ;    MSB of r24 to carry bit
```

  - Exclusive OR (XOR): if applied to same register then it clears the register.

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	⊥
⊥	⊥	⊥	⊥

Example: {1,1,0,1,⊥,0,1,0} AND {0, ⊥, ⊥,1,1,0, ⊥, ⊥} = {0, ⊥,0,1, ⊥,0, ⊥,0}

13

Dataflow Analysis

Managing abstract machine state

- Create a copy of abstract state at points of conditional instructions (if unable to determine)
- Question: when to merge two abstract states?
  - Too infrequently → resource & time consuming
  - Too frequently → imprecise results
  - ➲ Trade-off between resource & accuracy
- Criteria for merging states:
  - Context-sensitive & -insensitive: each invocation of same function is analyzed separately or not
  - Path-sensitive & -insensitive: each control-flow path is analyzed separately or not (always merge at joint points)
- Authors’ tool: path- and context-sensitive (more precise interrupt masks leading to more precise results)

14

Dataflow Analysis

Assumptions

- Indirect stores to memory do not modify return addresses and registers mapped into memory space.
- No self-modifying code (discouraged in embedded systems). If found, raise error.
- Explicit stack pointer modification: recognized by *macro instructions* (with observation that instructions changing SP come in several patterns). If not recognized, raise error.
- Indirect branches: indirect call, changing return address, e.g. in preemptive RTOS, is currently not considered. However, it is analogous to the case of interrupt handlers.
- Recursion: developers must explicitly specify maximum iteration count. Treated to maximum depth in analysis.
- Modeling stack frames: model the values on stack to improve accuracy (experimental)

15

Stack Depth Analysis

- Assumptions
  - *Acyclic IPG*: usually interrupt handlers are not re-entrant (i.e. at most one instance of it at one time), leading to acyclic IPG. If cyclic, max number of re-entrances must be specified.
  - *Machine state before interrupt is saved and is restored properly on return*
- Compute stack depth: traverse the (acyclic) IPG
  - `depth(i)`: stack depth bound of interrupt handler `i`
  - `depth(i,j)`: max stack depth of interrupt handler `i` at program points where interrupt handler `j` is enabled, or $-\infty$ if `j` is never enabled by `i`.
  - Worst-case stack depth (wcsd) of interrupt handler `i`

$$wcsd(i) = \max\{depth(i), \max_{j=1..n}\{depth(i,j)+wcsd(j)\}\}$$

  - Global stack depth bound is wcsd(0) of reset interrupt vector.

16

Evaluation of Technique & Tool

- Used simulator to validate the abstract interpreter
- Validating stack bounds: it must be true that

  worst-observed (testing-based) ≤ true worst ≤ estimated worst

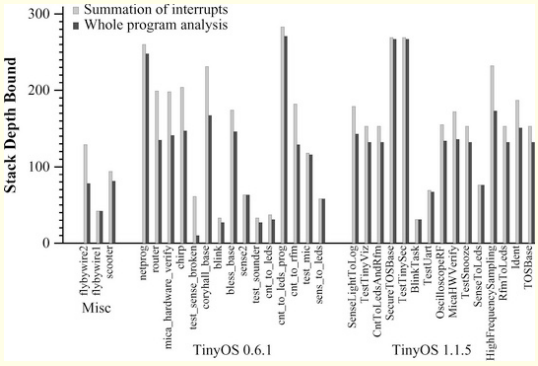
  Good stack bound result (from a TinyOS example):

Observed worst-case depths	Stack bound results
Int 0 = 27	Int 0 = 29
(not supported)	Int 1 = 2
Int 15 = 19	Int 15 = 19
Int 17 = 29	Int 17 = 29
(not supported)	Int 18 = 21
(not supported)	Int 20 = 23
Int 21 = 30	Int 21 = 30

The tool also provides bounds for interrupts not supported by the simulator (testing-based approach)

Evaluation of Technique & Tool

- The global analysis
  - Analyzed 75 embedded applications: TinyOS applications (old version with C, new version with nesC), several control programs (self-balancing scooter, autonomous helicopter)
  - Some results (figure)
  - Quite fast:
    - On Pen4, 3 GHz
    - Only 13 required more than 1s
    - Worst: 9s

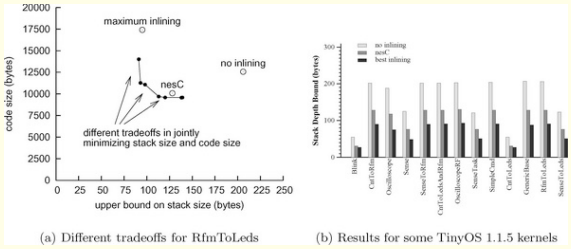


Reducing Stack Depth

- Useful to free memory, allow to use cheaper CPU.
- Utilize feedback information from the analysis tool.
- Way #1: reduce stack requirement of each edge in IPG
  - Currently implement function inlining
  - Use cost-function to balance between stack depth & code size
  - Rather effective but quite slow (80 minutes for an example!!!)
  - More advanced algorithms and support from compilation techniques may help
- Way #2: eliminating unnecessary preemption
  - Simplify the shape of the IPG by removing unnecessary edges
  - Developers explicitly specify edges to be removed (based on understanding of the system with the help of the tool).

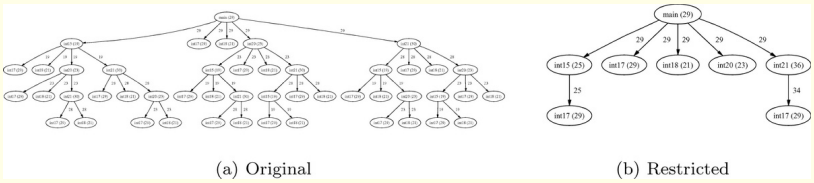
Reducing Stack Depth

- Example of way #1



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

- Example of way #2 (TinyOS's RfmToLeds example)



(a) Original (b) Restricted

Stack depth bound reduces from 128 bytes to 92 bytes

Conclusion

- Stack safety is important in embedded systems, but hard to verify by testing.
- Authors have developed techniques and experimental tools for static analysis of embedded software to find stack depth bound.
- Two novel methods have been developed for reducing stack memory requirement, utilizing the analysis tool.
- Design of the analyzer is more of an art than a science, with various tradeoffs.
- Source code:
  - Analyzer: http://www.cs.utah.edu/~regehr/stacktool/
  - Global inliner for GCC: http://www.cs.utah.edu/flux/knit/cmi.html

21

References

- J. Regehr, A. Reid, K. Webb. *Eliminating stack overflow by abstract interpretation.* ACM Trans. Emb. Comp. Sys., Nov 2005.
- J. Engblom. *Static properties of commercial embedded real-time programs, and their implication for worst-case execution time analysis.* IEEE Real-time Technology and Application Symp., 1999.
- D. Brylow, N. Damgaard, J. Palsberg. *Static checking of interrupt-driven software.* Intl. Conf. on Software Engineering, 2001.

22