# Eliminating Stack Overflow by Abstract Interpretation

**John Regehr**

**Alastair Reid**

**Kirk Webb**

**University of Utah**

# Contribution Preview

- **Evaluation of tradeoffs in stack depth analysis**
- **Automatic reduction of stack requirements**

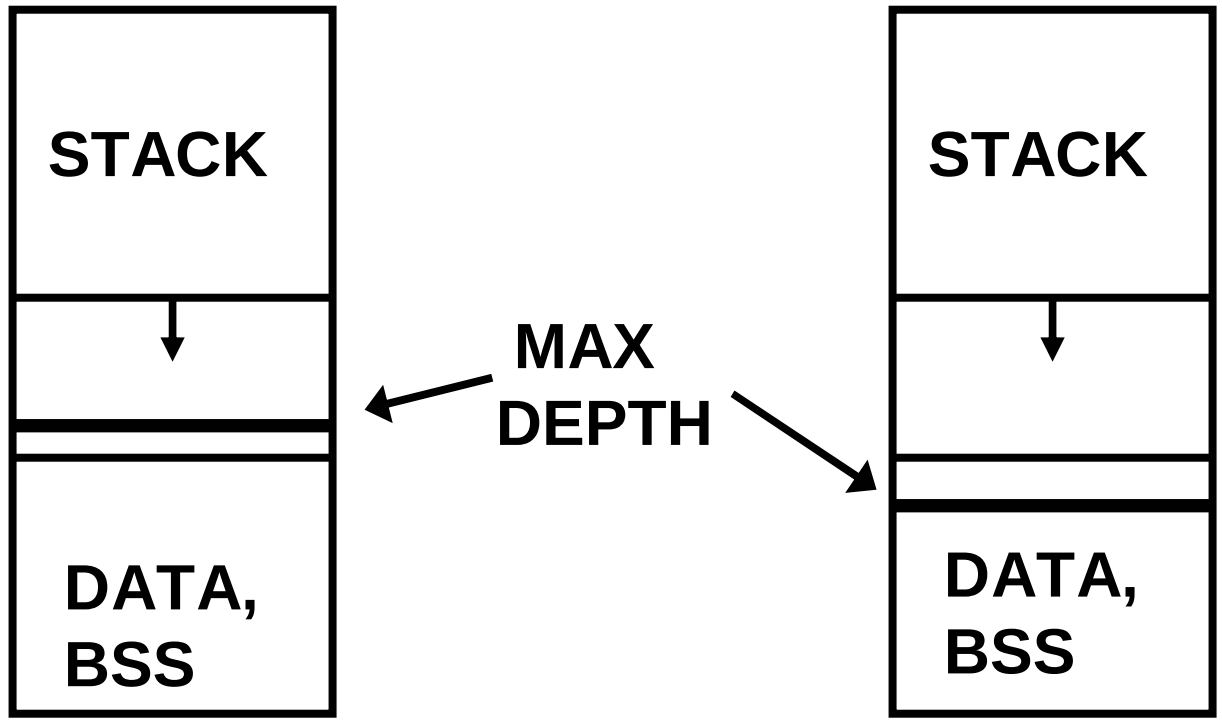
# Focus

- Whole program analysis…
- For legacy embedded systems in C and assembly…
- Running on microcontrollers
  - AVR, PIC, 68HC11, Z80, etc.
  - Typically limited to on-chip memory

# Goal: Bounded Stack Depth

- There are lots of stacks out there
- Stack region too big →
  - Wasted $$, power
- Stack region too small →
  - OOM exceptions or data corruption
- Stack region just right →
  - Safe, efficient system

# Using Stack Bounds




**SAFE** **UNSAFE**

# Analyze Source or Binary?

- Hard to predict compiler behavior
- Source for RTOS and libraries commonly not available
- Executable systems have no dangling references

# Analysis Overview

- Take a compiled and linked system

1. Compute approximate control flow graph
   - Tricky: indirect branches
2. Find “longest” path through the control flow graph
   - Tricky: recursion, loads into stack pointer, interrupts

# Interrupts

- Asynchronous events that may arrive when
  - Enable bit is set AND
  - Global enable bit is set
- Problem: Stack depth strongly affected by interrupt preemption relations
- Solution: Compute static estimate of interrupt mask at each program point
  - Then, use algorithm from Brylow et al. (ICSE 2001)

# Motivating Code

```
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
```

# Abstract Machine Model

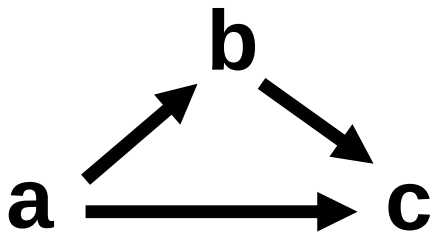
- **Storage locations contain tri-valued bits: 0, 1, $\perp$**
  - **{0,0,1,1} | {0,$\perp$,1,$\perp$} = {0,$\perp$,1,1}**
- **Only model registers**
  - **General-purpose + special purpose**
  - **Modeling main memory unnecessary and difficult**

# Analysis

- **Overview:**
  - **Initialize worklist with entry points**
    - **Reset, interrupts**
  - **Process worklist items until fixpoint is reached**
- **Has context insensitive and sensitive modes**
- **Implementation:**
  - **9000 lines of C (3000 generated)**

# Context (In)Sensitivity

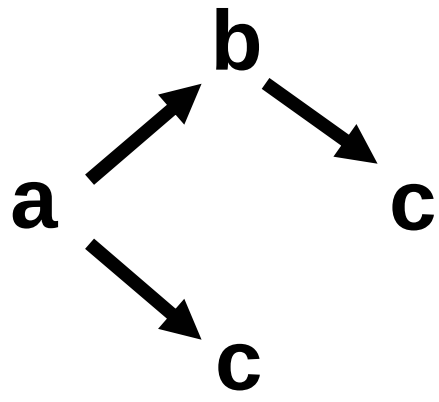
```
b() {
  c();
}
a() {
  b();
  c();
}
```




**Context insensitive**

# Context (In)Sensitivity

```
b() {
  c();
}
a() {
  b();
  c();
}
```




**Context sensitive**

# First Approach

- No abstract interpretation
  - Just add up requirements of individual interrupts
  - Assumption: at most one outstanding instance of each interrupt
- Cheap and easy
  - Can be written in ~400 lines of Perl!

# Second Approach

- Context insensitive analysis
- Large implementation effort (relative to first approach)
- Fast and memory-efficient

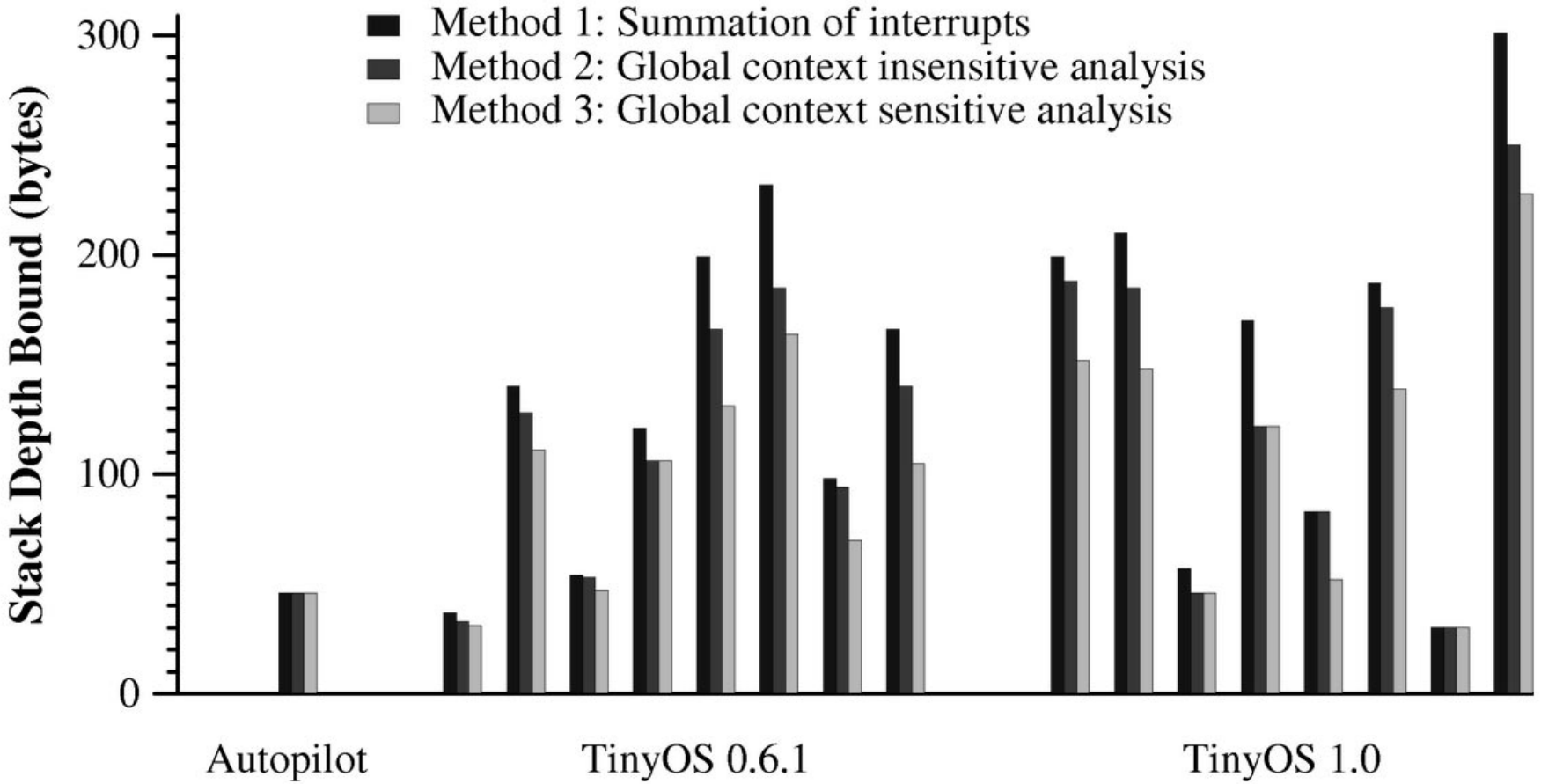
# Third Approach

- **Context sensitive analysis**
- **Small implementation effort (relative to second approach)**
- **Relatively slow and memory-intensive**
  - **Requires exponential space / time in the worst case**
  - **In practice analysis took at most 4 seconds and 140 MB**

# Test Programs

- 64 programs from TinyOS 0.6.1 and 1.0
- “flybywire” from the Autopilot project
- Up to 30,000 lines of C
- All run on Atmel ATMega 103
  - 8-bit architecture
  - 4 KB RAM, 128 KB flash

# Results


Method 1: Summation of interrupts
Method 2: Global context insensitive analysis
Method 3: Global context sensitive analysis
Stack Depth Bound (bytes)
300
200
100
0
Autopilot
TinyOS 0.6.1
TinyOS 1.0

# Validation

1. **Are machine states from actual runs “within” the static analysis?**
   - **Yes**
2. **What fraction of instructions have statically known int. mask?**
   - **Content insensitive: 59%**
   - **Context sensitive: 98%**
3. **Can we observe system using worst-case stack depth?**
   - **Usually, for simple examples**
   - **No, for complex programs**

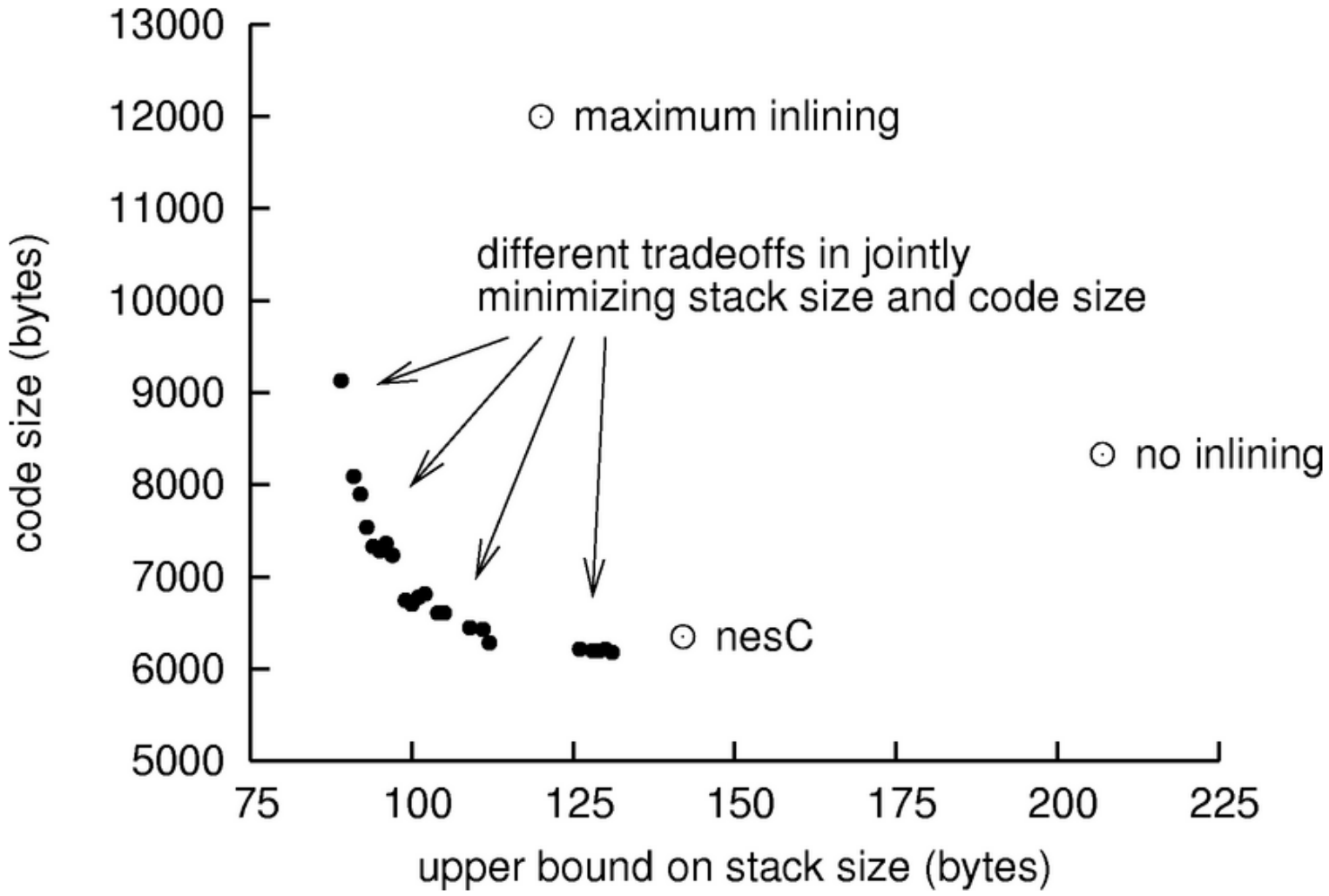
# Reducing Stack Size

- **Observation: Function inlining often decreases stack requirements**
  - **Avoids pushing registers, frame pointer, return address**
  - **Called code can be specialized**
- **Strategy: Use stack tool output as input to global inlining tool**

# Challenges

1. **Inlining causes code bloat**
   - **Solution: Minimize user-defined cost function that balances stack memory and code size**
2. **Inlining sometimes increases stack depth!**
   - **Solution: Trial compilations**
3. **Search space is exponential in number of static function calls**
   - **Solution: Heuristic search**

# Results


13000
12000
11000
10000
9000
8000
7000
6000
5000
code size (bytes)
maximum inlining
different tradeoffs in jointly
minimizing stack size and code size
no inlining
nesC
75
100
125
150
175
200
225
upper bound on stack size (bytes)

# Related Projects

- **Bounding stack depth:**
  - **Brylow et al., ICSE 01**
  - **Chatterjee et al., SAS 03**
  - **StackAnalyzer from Absint**
- **Automatically reducing stack depth:**
  - **No related work that we know of!**
- **Observation: Bounding and minimizing memory use is a pretty open research area**

# Future Work

- Multi-objective optimization
  - E.g. stack depth + code size + code speed
- More uses for abs. int.
  - WCET estimation
  - Bug finding
- Support more chips
  - Currently only Atmel AVR family
  - Working on automating this

# Summary of Contributions

- **Evaluated tradeoffs in analysis complexity**
  - **35% tighter bounds for context sensitive**
- **Effective automatic stack depth reduction**
  - **32% lower stack requirement compared to a smart non-stack-oriented inlining policy**

**More info and papers here:**

**http://www.cs.utah.edu/~regehr/**

**Stack tool code available soon!**