Static Analysis of Worst-Case Stack Cache Behavior

Alexander Jordan[1] Florian Brandner[2]
Martin Schoeberl[2]

Institute of Computer Languages[1]
Compiler and Languages Group
Vienna University of Technology
ajordan@complang.tuwien.ac.at

Embedded Systems Engineering Section[2]
DTU Compute
Technical University of Denmark
{flbr,masca}@imm.dtu.dk

T-CREST

Motivation

- Caches add complexity to WCET analysis
- Mitigation strategies (by design):
  - Separate caches
  - Adapt cache to access patterns

Introduction

Motivation

- Caches add complexity to WCET analysis
- Mitigation strategies (by design):
  - Separate caches
  - Adapt cache to access patterns

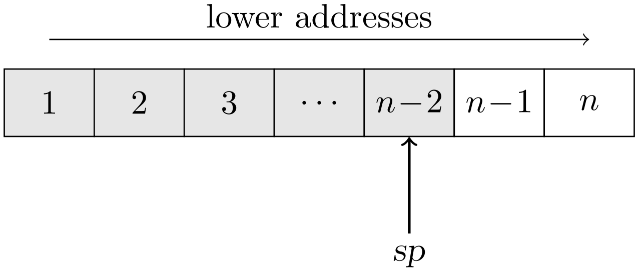
A Cache For Stack Data

- Suits the program's execution stack
- Stack access (load, store) always hits the cache
- Special instructions control the stack
  - May trigger load/store from/to main memory (delays)

Stack Cache Introduction

Logical View

Stack cache with *n* blocks (2 available)



Stack Cache Introduction

```
func A:
    sres 2;
    store #1;
    B();
    sens 2;
    load #1;
    C();
    sens 2;
    sfree 2;
end;
```

Reserve

allocates *k* blocks in the stack cache

- spills minimal number of blocks if cache capacity exceeded

Stack Cache Introduction

```
func A:
    sres 2;
    store #1;
    B();
    sens 2;
    load #1;
    C();
    sens 2;
    sfree 2;
end;
```

Free

discards *k* most recently reserved blocks

Stack Cache Introduction

```
func A:
  sres 2;
  store #1;
  B();
  sens 2;
  load #1;
  C();
  sens 2;
  sfree 2;
end;
```

Ensure

if not all *k* blocks of the current frame are available in the cache

- fills cache with missing blocks

Stack Cache Introduction

```
func A:
    sres 2;
    store #1;
    B();
    sens 2;
    load #1;
    C();
    sens 2;
    sfree 2;
end;
```

Ensure

if not all *k* blocks of the current frame are available in the cache

- fills cache with missing blocks
- can prevent loading of redundant values

Stack Cache Introduction

```
func A:
    sres 2;
    store #1;
    B();
    sens 2;
    load #1;
    C();
    sens 2;
    sfree 2;
end;
```

Ensure

if not all *k* blocks of the current frame are available in the cache

- fills cache with missing blocks
- can prevent loading of redundant values

Analysis Problem

Two Problems

- Worst-case filling of ensure instructions (Ensure Analysis)
- Worst-case spilling of reserve instructions (Reserve Analysis)

Analysis Problem

Two Problems

- Worst-case filling of ensure instructions (Ensure Analysis)
- Worst-case spilling of reserve instructions (Reserve Analysis)

Analysis Goal

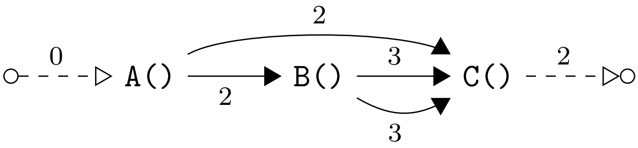
Find better bounds than arguments k

Analysis Foundations

Annotated Call Graph

Call graph with weights representing reserved stack space, including an artificial source and sink nodes.

Example: Annotated CG for 3 Functions



Occupancy

Fill-level of the stack cache

- Occupancy bounds (upper/lower)

Analysis Foundations

Occupancy

Fill-level of the stack cache

- Occupancy bounds (upper/lower)

Displacement

Data potentially evicted from stack cache during function call

- Minimum/maximum displacements

Analysis Foundations

Occupancy

Fill-level of the stack cache

- Occupancy bounds (upper/lower)

Displacement

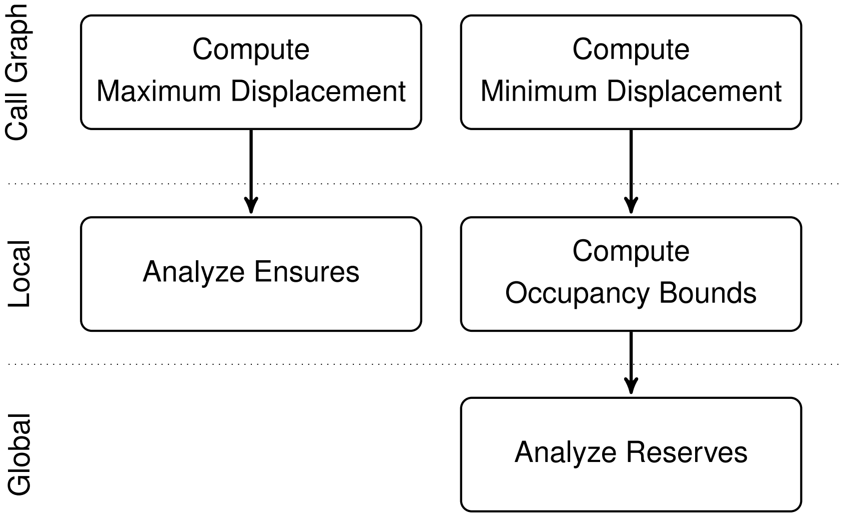
Data potentially evicted from stack cache during function call

- Minimum/maximum displacements

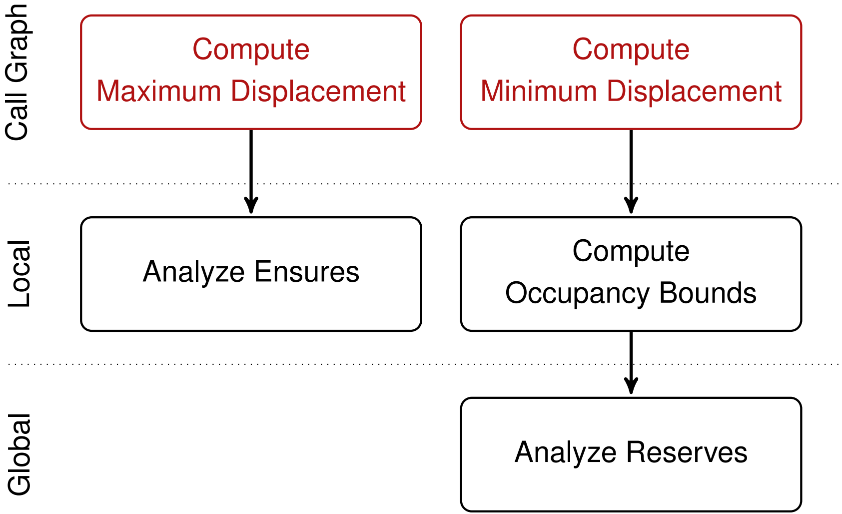
Context-Sensitivity

Analysis information for a program point depends on its call nesting (hierarchy of calling functions)

Analysis Algorithm



Analysis Algorithm



Computing Displacement

Displacement of a Call Site

Computed on the annotated call graph between call destination and sink node

- Minimum displacement: shortest path search
- Maximum displacement: longest path search

Computing Displacement

Displacement of a Call Site

Computed on the annotated call graph between call destination and sink node

- Minimum displacement: shortest path search
- Maximum displacement: longest path search

Acyclic Call Graphs

Easy to compute with dynamic programming

Computing Displacement

Displacement of a Call Site

Computed on the annotated call graph between call destination and sink node

- Minimum displacement: shortest path search
- Maximum displacement: longest path search

Acyclic Call Graphs

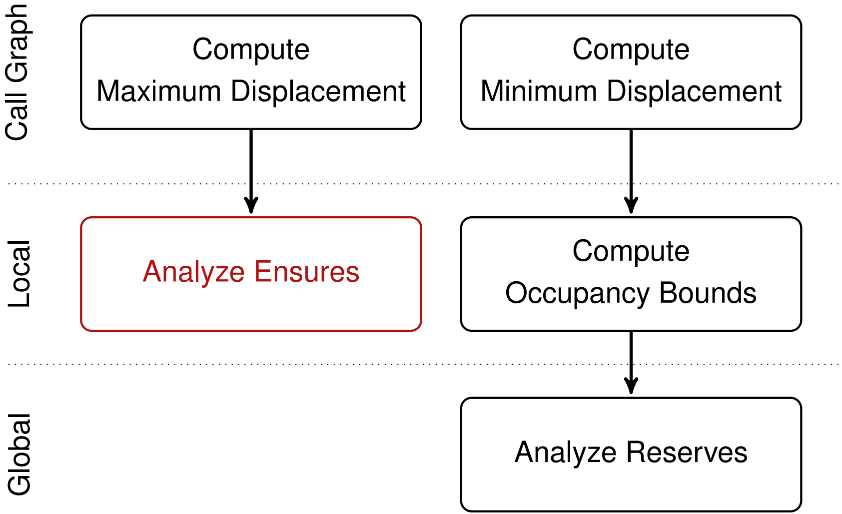
Easy to compute with dynamic programming

Call Graphs With Recursion

Can be modeled using an ILP

- In fact: shortest (longest) tail in the call graph
- Allows (user) bounds for program's calling behavior

Analysis Algorithm



Ensure Analysis

- Input: maximum displacement
- Output: worst-case filling value for every ensure
- context-insensitive result and analysis

Ensure Analysis

Ensure Analysis

- Input: maximum displacement
- Output: worst-case filling value for every ensure
- context-insensitive result and analysis

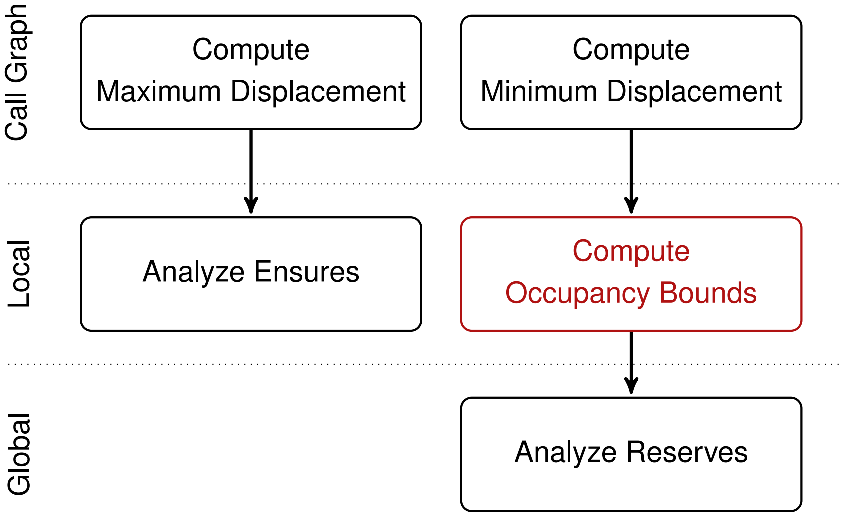
Function-local computation

Worst-case filling only depends on

- space reserved at function entry (static)
- minimum occupancy (induced by maximum displacement) of all paths reaching the ensure

Thus can be solved by local data flow analysis.

Analysis Algorithm



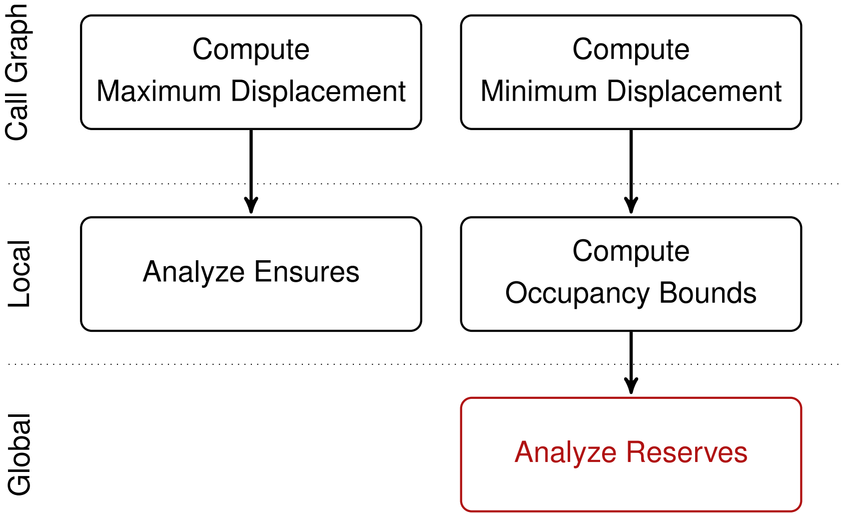
Computing Occupancy Bounds

Maximum Occupancy of a Call Site

Inverse to minimum occupancy used by ensure analysis

- Solved the same way: local data-flow analysis, but
- use the minimum displacement
- assume full stack cache at function entry

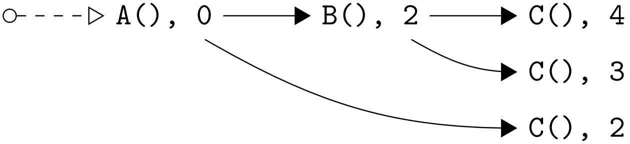
Analysis Algorithm



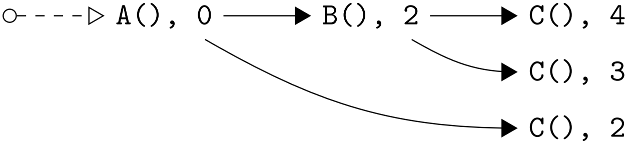
Reserve Analysis

- Input: annotated call graph, occupancy bounds
- Output: spill cost graph (stack-context-sensitive)
- Starting with initially empty stack cache, derive new cache contexts from the annotated call graph
- Occupancy bounds limit the number of distinct contexts that need to be propagated

Example: Spill Cost Graph



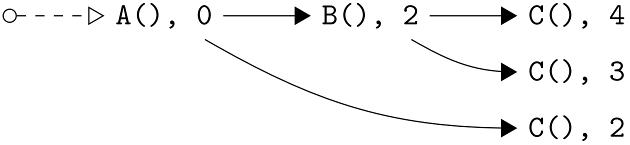
Example: Spill Cost Graph



Spill cost derived from graph: $\widehat{c}_s \cdot \max(0, o + k - |SC|)$

Reserve Analysis

Example: Spill Cost Graph



Spill cost derived from graph: $\widehat{c}_s \cdot \max(0, o + k - |SC|)$

Spill Cost Graph Pruning Opportunities

- Contexts of the same function with 0 spill cost can be merged
- Infeasible contexts (user bounds) can be pruned
- Possible trade-off: analysis precision vs. graph size

Results

Evaluation

- Platform: Patmos (LLVM compiler)
- Benchmarks: MiBench
- Several stack cache sizes

Results

Evaluation

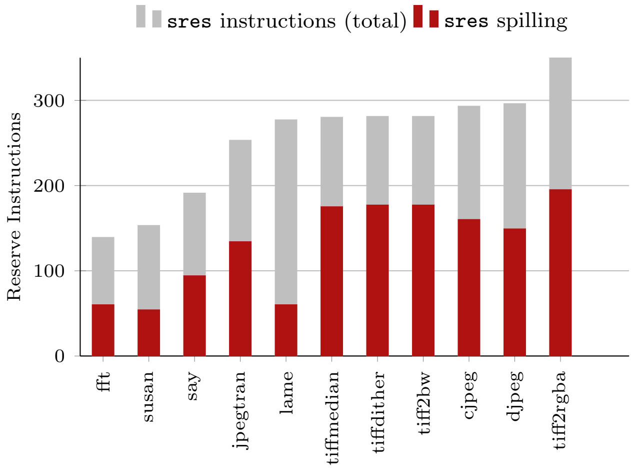
- Platform: Patmos (LLVM compiler)
- Benchmarks: MiBench
- Several stack cache sizes

Analysis Overhead

- Up to 94 ILPs
- 1.30s average analysis time
- Up to 53487 nodes in spill cost graph
  - Reduced to 17254 by pruning

Results

Spilling Reserves



Conclusion

Worst-case stack cache analysis

- Efficient analysis
  - Separate analysis problems
  - Performed at different levels
- Computed through
  - Augmented path search
  - Data-flow analysis
- Analysis results
  - Context-sensitive where required
  - Spill-cost graph precision can be lowered on demand
  - Ready for use in WCET tool