Static Analysis of Worst-Case Stack Cache Behavior

Alexander Jordan¹  Florian Brandner²
Martin Schoeberl²

Institute of Computer Languages¹
Compiler and Languages Group
Vienna University of Technology
ajordan@complang.tuwien.ac.at

Embedded Systems Engineering Section²
DTU Compute
Technical University of Denmark
{flbr,masca}@imm.dtu.dk
Motivation

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

Stack cache with \( n \) blocks (2 available)

\[
\begin{array}{cccccccc}
1 & 2 & 3 & \cdots & n-2 & n-1 & n \\
\end{array}
\]

\( sp \)
func A:
1  sres 2;
2  store #1;
3  B();
4  sens 2;
5  load #1;
6  C();
7  sens 2;
8  sfree 2;
end;

Reserve
allocates $k$ blocks in the stack cache

- spills minimal number of blocks if cache capacity exceeded
func A:
1  sres 2;
2  store #1;
3  B();
4  sens 2;
5  load #1;
6  C();
7  sens 2;
8  sfree 2;
end;

Free

discards $k$ most recently reserved blocks
func A:
1  sres 2;
2  store #1;
3  B();
4  sens 2;
5  load #1;
6  C();
7  sens 2;
8  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

```plaintext
func A:
1  sres 2;
2  store #1;
3  B();
4  sens 2;
5  load #1;
6  C();
7  sens 2;
8  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
func A:
1  sres 2;
2  store #1;
3  B();
4  sens 2;
5  load #1;
6  C();
7  sens−2;
8  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
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 $k$
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

Call Graph

- Compute Maximum Displacement
- Compute Minimum Displacement

Local

- Analyze Ensures
- Compute Occupancy Bounds

Global

- Analyze Reserves
Analysis Algorithm

Call Graph
- Compute Maximum Displacement
- Compute Minimum Displacement

Local
- Analyze Ensures
- Compute Occupancy Bounds

Global
- Analyze Reserves
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
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

Call Graph

- Compute Maximum Displacement
- Compute Minimum Displacement

Local

- Analyze Ensures
- Compute Occupancy Bounds

Global

- Analyze Reserves
Ensure Analysis

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

**Call Graph**
- Compute Maximum Displacement
- Compute Minimum Displacement

**Local**
- Analyze Ensures
- Compute Occupancy Bounds

**Global**
- Analyze Reserves
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

Call Graph
- Compute Maximum Displacement
- Compute Minimum Displacement

Local
- Analyze Ensures
- Compute Occupancy Bounds

Global
- Analyze Reserves
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
Reserve Analysis

Example: Spill Cost Graph

0 \rightarrow A(), 0 \rightarrow B(), 2 \rightarrow C(), 4
\rightarrow C(), 3
\rightarrow C(), 2
Reserve Analysis

Example: Spill Cost Graph

A(), 0 → B(), 2 → C(), 4

C(), 3

C(), 2

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

Example: Spill Cost Graph

\[ O \rightarrow A(\cdot), 0 \rightarrow B(\cdot), 2 \rightarrow C(\cdot), 4 \]
\[ \rightarrow C(\cdot), 3 \]
\[ \rightarrow C(\cdot), 2 \]

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

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
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

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

![Bar chart showing spilling reserves for various programs]

- **sres instructions (total)**
- **sres spilling**

Programs:
- fft
- susan
- say
- jpegtran
- lame
- tiffmedian
- tiffdither
- tiff2bw
- cjpeg
- djpeg
- tiff2rgba

Reserve Instructions:
- 0
- 100
- 200
- 300

Alexander Jordan, Florian Brandner, Martin Schoeberl
Static Analysis of Worst-Case Stack Cache Behavior
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