Static Analysis of Worst-Case Stack Cache Behavior

Florian Brandner
Unité d’Informatique et d’Ing. des Systèmes
ENSTA-ParisTech

Alexander Jordan
Embedded Systems Engineering Sect.
Technical University of Denmark

This work is partially supported by the EC project T-CREST.
Real-Time Systems

Strict timing guarantees

• Critical tasks have to be completed in time
Real-Time Systems

**Strict timing guarantees**

- Critical tasks have to be completed in time
- Bound *Worst-Case Execution Time* (WCET)
WCET Analysis

Bound longest possible execution time of a program
• Covering all potential execution paths
• Covering all potential program inputs
• Covering all potential hardware states
Bound longest possible execution time of a program

- Covering all potential execution paths
- Covering all potential program inputs
- Covering all potential hardware states
  - Processor pipeline
  - Branch predictors
  - Data and instruction caches
  - Main memory
Example: Miss/Hit Classification

Initial cache state*

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x101</td>
<td>0x103</td>
</tr>
</tbody>
</table>

*Cache configuration
2-way set-associative, 1 word blocks, 2 cache lines, LRU replacement
Example: Miss/Hit Classification

Initial cache state

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x101</td>
<td>0x103</td>
</tr>
</tbody>
</table>

lw [0x100]  

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x101</td>
<td>0x103</td>
</tr>
</tbody>
</table>

Classified as hit

*Cache configuration
2-way set-associative, 1 word blocks, 2 cache lines, LRU replacement
Example: Miss/Hit Classification

Initial cache state*

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x101</td>
<td>0x103</td>
</tr>
</tbody>
</table>

lw [0x100]

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x101</td>
<td>0x103</td>
</tr>
</tbody>
</table>

Classified as hit

lw [0x105]

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x105</td>
<td>0x101</td>
</tr>
</tbody>
</table>

Classified as miss

*Cache configuration

2-way set-associative, 1 word blocks, 2 cache lines, LRU replacement
Example: Miss/Hit Classification

Initial cache state*

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x101</td>
<td>0x103</td>
</tr>
</tbody>
</table>

lw [0x100]  

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x101</td>
<td>0x103</td>
</tr>
</tbody>
</table>

Classified as hit

lw [0x105]  

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x105</td>
<td>0x101</td>
</tr>
</tbody>
</table>

Classified as miss

lw [??]  

<table>
<thead>
<tr>
<th>??</th>
<th>??</th>
</tr>
</thead>
<tbody>
<tr>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>

Classification unclear

*Cache configuration
2-way set-associative, 1 word blocks, 2 cache lines, LRU replacement
Example: Miss/Hit Classification

Initial cache state

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x200</td>
</tr>
<tr>
<td>0x101</td>
<td>0x103</td>
</tr>
</tbody>
</table>

\[\text{lw} \ [0x100]\]

\[
\begin{array}{cc}
0x100 & 0x200 \\
0x101 & 0x103 \\
\end{array}
\]

Classified as hit

\[\text{lw} \ [0x105]\]

\[
\begin{array}{cc}
0x100 & 0x200 \\
0x105 & 0x101 \\
\end{array}
\]

Classified as miss

\[\text{lw} \ [??]\]

\[
\begin{array}{cc}
?? & ?? \\
?? & ?? \\
\end{array}
\]

Classification unclear

Main challenge
The abstract cache state of the analysis depends on the precise address and order of the executed memory accesses.

*Cache configuration
2-way set-associative, 1 word blocks, 2 cache lines, LRU replacement
Context-Sensitivity

Miss/hit classification requires

- Precise information to disambiguate addresses
- High levels of context-sensitivity
- High levels of virtual loop unrolling
- Analysis effort is multiplied accordingly
Context-Sensitivity

Miss/hit classification requires

- Precise information to disambiguate addresses
- High levels of context-sensitivity
- High levels of virtual loop unrolling
- Analysis effort is multiplied accordingly

Main problem

Subsequent phases of WCET analysis suffer from high complexity due to this virtual code duplication.
Alternative Solution

Predictable caching

- Dedicated caches designed for analyzability/predictability
- Easy to analyze
- Simple hardware design
- Requiring no/little information on accesses addresses
Alternative Solution

Predictable caching

• Dedicated caches designed for analyzability/predictability
• Easy to analyze
• Simple hardware design
• Requiring no/little information on accesses addresses

In this work
Time-predictable caching of stack data using a stack cache.
What is a Stack Cache?

Dedicated cache for stack data

• Simple ring buffer (*FIFO replacement*)
• All stack accesses are guaranteed hits (no need to analyze them)
• Dedicated stack control instructions (need to be analyzed)
  • sres x: reserve x blocks on the stack
  • sfree x: free x blocks on the stack
  • sens x: ensure that at least x blocks are cached

• Intuitively: a cache window following the stack top
Example: Stack Cache

<table>
<thead>
<tr>
<th>(1) function A()</th>
<th>function B()</th>
<th>function C()</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2) sres 2</td>
<td>sres 3</td>
<td>sres 2</td>
</tr>
<tr>
<td>(3) call B()</td>
<td>call C()</td>
<td></td>
</tr>
<tr>
<td>(4) sens 2</td>
<td>sens 3</td>
<td></td>
</tr>
<tr>
<td>(5) call C()</td>
<td>call C()</td>
<td></td>
</tr>
<tr>
<td>(6) sens 2</td>
<td>sens 3</td>
<td></td>
</tr>
<tr>
<td>(7) sfree 2</td>
<td>sfree 3</td>
<td></td>
</tr>
</tbody>
</table>

Logical stack

Stack cache*

*Cache configuration: 4 blocks
Example: Stack Cache

(1) function A() function B() function C()
(2) sres 2 ← sres 3 sres 2
(3) call B() call C() sfree 2
(4) sens 2 sens 3
(5) call C() call C()
(6) sens 2 sens 3
(7) sfree 2 sfree 3

Logical stack

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
</tr>
</thead>
</table>

Stack cache*

| A | A |

*Cache configuration: 4 blocks
Example: Stack Cache

(1) function A() function B() function C()
(2) sres 2 sres 3 sres 2
(3) call B() ← call C() sfree 2
(4) sens 2 sens 3 sens 2
(5) call C() call C() call C()
(6) sens 2 sens 3 sens 3
(7) sfree 2 sfree 3 sfree 3

Logical stack

Stack cache*

*Cache configuration: 4 blocks
Example: Stack Cache

(1) function A() function B() function C()
(2) sres 2 sres 3 ← sres 2
(3) call B() call C() sfree 2
(4) sens 2 sens 3
(5) call C() call C()
(6) sens 2 sens 3
(7) sfree 2 sfree 3

Logical stack

Stack cache*

*Cache configuration: 4 blocks
Example: Stack Cache

(1) function A()  
(2) sres 2  
(3) call B()  
(4) sens 2  
(5) call C()  
(6) sens 2  
(7) sfree 2

function B()  
sres 3  
call C()  
sens 3  
call C()  
sens 3  
sfree 2

Logical stack

Stack cache*

*Cache configuration: 4 blocks
Example: Stack Cache

(1) function A() function B() function C()
(2) sres 2 sres 3 sres 2
(3) call B() call C() sfree 2
(4) sens 2 sens 3
(5) call C() call C()
(6) sens 2 sens 3
(7) sfree 2 sfree 3

Logical stack

```
A | A | B | B | B | C | C
```

Stack cache*

```
B | B | C | C
```

spill 2 blocks

*Cache configuration: 4 blocks
Example: Stack Cache

1. function A()  
2. sres 2  
3. call B()  
4. sens 2  
5. call C()  
6. sens 2  
7. sfree 2  

Logical stack:

```
A   A   B   B   B
```

Stack cache:

```
B   B
```

*Cache configuration: 4 blocks*
Example: Stack Cache

(1) function A()  function B()  function C()
(2) sres 2       sres 3       sres 2
(3) call B()      call C()     sfree 2
(4) sens 2       sens 3 ←
(5) call C()      call C()     sfree 3
(6) sens 2       sens 3
(7) sfree 2       sfree 3

Logical stack

| A | A | B | B | B |   |   |   |

Stack cache*

| B | B | B |   | fill 1 block |

*Cache configuration: 4 blocks
Example: Stack Cache

(1) function A() function B() function C()
(2) sres 2 sres 3 sres 2
(3) call B() call C() sfree 2
(4) sens 2 sens 3
(5) call C() call C() ←
(6) sens 2 sens 3
(7) sfree 2 sfree 3

Logical stack

Stack cache*

*Cache configuration: 4 blocks
Example: Stack Cache

(1) function A() function B() function C()
(2) sres 2 sres 3 sres 2 ←
(3) call B() call C() sfree 2
(4) sens 2 sens 3
(5) call C() call C()
(6) sens 2 sens 3
(7) sfree 2 sfree 3

Logical stack

| A | A | B | B | B | C | C |

Stack cache*

| B | B | C | C | spill 1 block |

*Cache configuration: 4 blocks
### Example: Stack Cache

<table>
<thead>
<tr>
<th>Line</th>
<th>Code 1</th>
<th>Code 2</th>
<th>Code 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>function A()</td>
<td>function B()</td>
<td>function C()</td>
</tr>
<tr>
<td>(2)</td>
<td>sres 2</td>
<td>sres 3</td>
<td>sres 2</td>
</tr>
<tr>
<td>(3)</td>
<td>call B()</td>
<td>call C()</td>
<td>sfree 2 ←</td>
</tr>
<tr>
<td>(4)</td>
<td>sens 2</td>
<td>sens 3</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>call C()</td>
<td>call C()</td>
<td></td>
</tr>
<tr>
<td>(6)</td>
<td>sens 2</td>
<td>sens 3</td>
<td></td>
</tr>
<tr>
<td>(7)</td>
<td>sfree 2</td>
<td>sfree 3</td>
<td></td>
</tr>
</tbody>
</table>

**Logical stack**

| A | A | B | B | B |

**Stack cache**

| B | B |

*Cache configuration: 4 blocks*
Example: Stack Cache

1. `function A()`
2. `sres 2`
3. `call B()`
4. `sens 2`
5. `call C()`
6. `sens 2`
7. `sfree 2`

Logical stack:

```
A A B B B
```

Stack cache:

```
B B B
```

*Cache configuration: 4 blocks*
Example: Stack Cache

(1) function A() function B() function C()
(2) sres 2 sres 3 sres 2
(3) call B() call C() sfree 2
(4) sens 2 sens 3
(5) call C() call C()
(6) sens 2 sens 3
(7) sfree 2 sfree 3 ←

Logical stack

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stack cache*

*Cache configuration: 4 blocks
### Example: Stack Cache

<table>
<thead>
<tr>
<th></th>
<th>function A()</th>
<th>function B()</th>
<th>function C()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>sres 2</td>
<td>sres 3</td>
<td>sres 2</td>
</tr>
<tr>
<td>3</td>
<td>call B()</td>
<td>call C()</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>sens 2 ←</td>
<td>sens 3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>call C()</td>
<td>call C()</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>sens 2</td>
<td>sens 3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>sfree 2</td>
<td>sfree 3</td>
<td></td>
</tr>
</tbody>
</table>

**Logical stack**

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
</tr>
</thead>
</table>

**Stack cache**

| A | A |   |   |   |

*Fill 2 blocks*

*Cache configuration: 4 blocks*
Example: Stack Cache

<table>
<thead>
<tr>
<th></th>
<th>function A()</th>
<th>function B()</th>
<th>function C()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>sres 2</td>
<td>sres 3</td>
<td>sres 2</td>
</tr>
<tr>
<td>3</td>
<td>call B()</td>
<td>call C()</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>sens 2</td>
<td>sens 3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>call C() ←</td>
<td>call C()</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>sens 2</td>
<td>sens 3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>sfree 2</td>
<td>sfree 3</td>
<td></td>
</tr>
</tbody>
</table>

**Logical stack**

```
A  A
```

**Stack cache***

```
A  A
```

*Cache configuration: 4 blocks*
Example: Stack Cache

(1) function A()  function B()  function C()
(2) sres 2        sres 3        sres 2 ←
(3) call B()      call C()      sfree 2
(4) sens 2        sens 3
(5) call C()      call C()      sfree 2
(6) sens 2        sens 3
(7) sfree 2       sfree 3

Logical stack

\[
\begin{array}{ccccccc}
A & A & C & C & \ldots & & \\
\end{array}
\]

Stack cache*

\[
\begin{array}{cccc}
A & A & C & C \\
\end{array}
\]

*Cache configuration: 4 blocks
Example: Stack Cache

(1) function A()  function B()  function C()  
(2) sres 2  sres 3  sres 2  
(3) call B()  call C()  
(4) sens 2  sens 3  
(5) call C()  call C()  
(6) sens 2  sens 3  
(7) sfree 2  sfree 3  

Logical stack

|   | A | A |   |   |   |   |   |

Stack cache*

|   | A | A |

*Cache configuration: 4 blocks
Example: Stack Cache

(1) function A()  function B()  function C()
(2) sres 2  sres 3  sres 2
(3) call B()  call C()  sfree 2
(4) sens 2  sens 3
(5) call C()  call C()  
(6) sens 2  sens 3
(7) sfree 2  sfree 3

Logical stack

| A | A |

Stack cache*

| A | A |

*Cache configuration: 4 blocks
Example: Stack Cache

(1) function A()  
(2) sres 2  
(3) call B()  
(4) sens 2  
(5) call C()  
(6) sens 2  
(7) sfree 2 ←

Logical stack

```
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
```

Stack cache*

```
[ ] [ ] [ ] [ ]
```

*Cache configuration: 4 blocks
Stack Cache Analysis

Two analysis problems

- Bound the maximum amount of *spilling* at *sres*-instructions
- Bound the maximum amount of *filling* at *sens*-instructions
- Other instructions have no impact (*sfree*, *loads*, *stores*)
Stack Cache Analysis

Two analysis problems
- Bound the maximum amount of *spilling* at sres-instructions
- Bound the maximum amount of *filling* at sens-instructions
- Other instructions have no impact (sfree, loads, stores)

Main task
Determine the maximum/minimum occupancy-level of the stack cache before sres/sens-instructions respectively.

*Assuming sres/sfree at function entry/exit and sens after function calls.*
Terminology

**Occupancy**
Number of cache blocks utilized at a given program point.

**Displacement**
Number of cache blocks spilled to main memory at a function call.
Terminology

**Occupancy**
Number of cache blocks utilized at a given program point.

**Displacement**
Number of cache blocks spilled to main memory at a function call.

**Observation**
Knowing an occupancy bound at function entry, the occupancy at a program point within that function can be bounded using the displacement of function calls on all paths to the program point.
Example: Displacement

(1) function A()  function B()  function C()
(2) sres 2       sres 3       sres 2
(3) call B()      call C()     sfree 2
(4) sens 2       sens 3
(5) call C()      call C()     
(6) sens 2       sens 3
(7) sfree 2      sfree 3

Displacement at call C(): 2
Example: Displacement

(1) function A()  function B()  function C()

(2) sres 2  sres 3  sres 2

(3) call B()  call C()  sfree 2

(4) sens 2  sens 3

(5) call C()  call C()

(6) sens 2  sens 3

(7) sfree 2  sfree 3

Displacement at call C(): 2
Displacement at call B(): 3 + 2 = 5
Example: Occupancy

(1) function A() function B() function C()
(2) sres 2 sres 3 sres 2
(3) call B() call C() sfree 2
(4) sens 2 sens 3
(5) call C() call C()
(6) sens 2 sens 3
(7) sfree 2 sfree 3

Occupancy at C()
A()₃ → B()₃ → C(): 4 → spill 2 blocks
Example: Occupancy

(1) function A() function B() function C()
(2) sres 2 sres 3 sres 2
(3) call B() call C() sfree 2
(4) sens 2 sens 3 sens 3
(5) call C() call C() sens 3
(6) sens 2 sfree 3
(7) sfree 2 sfree 3

Occupancy at C()

A()₃ → B()₃ → C(): 4 → spill 2 blocks
A()₃ → B()₅ → C(): 3 → spill 1 blocks
Example: Occupancy

(1) function A() function B() function C()
(2) sres 2 sres 3 sres 2
(3) call B() call C()
(4) sens 2 sens 3
(5) call C() call C()
(6) sens 2 sens 3
(7) sfree 2 sfree 3

Occupancy at C()

A(3) → B(3) → C(): 4 → spill 2 blocks
A(3) → B(5) → C(): 3 → spill 1 blocks
A(5) → C(): 2 → no spilling
Bounding the Displacement

Search path with shortest/longest tail on a weighted call graph

- **Edge weights**
  The number of blocks reserved in the calling function

- **Edges to artificial sink node**
  Capturing paths through functions without a call

Example

```
A()  B()  C()
0  2  2  3  3  2
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>A()</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>B()</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>C()</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Handling Recursion

Unbounded longest paths/tails

- Modeled using Integer Linear Programming (ILP)
- Recursion bounded by (user-supplied) ILP constraints
- Model nested calls as flow in/out of call graph nodes
Ensure Analysis

Bound the maximum filling at sens-instructions

1. Pre-compute the *maximum* displacement at function calls
2. Perform a function-local data-flow analysis
   - Propagate the minimum occupancy to sens-instructions
   - Assume a full stack cache at function entry
   - Adjust occupancy at sens-instructions
   - Adjust occupancy at call sites using maximum displacement
3. Bound worst-case filling using the minimum occupancy
Ensure Analysis

Bound the maximum filling at $s\alpha s$-instructions

1. Pre-compute the \textit{maximum} displacement at function calls
2. Perform a function-local data-flow analysis
   - Propagate the minimum occupancy to $s\alpha s$-instructions
   - Assume a full stack cache at function entry
   - Adjust occupancy at $s\alpha s$-instructions
   - Adjust occupancy at call sites using maximum displacement
3. Bound worst-case filling using the minimum occupancy

\textbf{Observation}

This analysis is context-insensitive.
Example: Ensure Analysis

```
function A()
sres 2
call B()
sens 2
call C()
sens 2
sfree 2
4
disp(B) = 5
0
Stack cache full
Stack cache empty
```
Example: Ensure Analysis

function A()
sres 2
call B()
sens 2
call C()
sens 2
sfree 2

Fill 2 blocks

16/27
Example: Ensure Analysis

function A()
sres 2
call B()
sens 2
call C()
sens 2
sfree 2
2
disp(C) = 2
Example: Ensure Analysis

function A()
sres 2
call B()
sens 2
call C()
sens 2
sfree 2

No filling
Reserve Analysis

Bound the maximum spilling at $s_{res}$-instructions

1. Pre-compute the $\textit{minimum}$ displacement at function calls
2. Perform a function-local data-flow analysis
   - Propagate the worst-case occupancy to function calls
   - Assume a full stack cache at function entry
   - Adjust occupancy at $s_{es}$-instructions
   - Adjust occupancy at call sites using minimum displacement
3. Perform an inter-procedural analysis on the call graph
   - Construct the $\textit{Spill Cost Graph}$
   - Iteratively discover new stack cache contexts
   - Derive new contexts by either
     • 3.1 Adding the amount of locally reserved blocks
     • 3.2 Propagating the worst-case occupancy
Reserve Analysis

Bound the maximum spilling at \texttt{sres}-instructions

1. Pre-compute the \textit{minimum} displacement at function calls
2. Perform a function-local data-flow analysis
   - Propagate the worst-case occupancy to function calls
   - Assume a full stack cache at function entry
   - Adjust occupancy at \texttt{sens}-instructions
   - Adjust occupancy at call sites using minimum displacement
3. Perform an inter-procedural analysis on the call graph
   - Construct the \textit{Spill Cost Graph}
   - Iteratively discover new stack cache contexts
   - Derive new contexts by either
     3.1 Adding the amount of locally reserved blocks
     3.2 Propagating the worst-case occupancy

\textbf{Observation}

The analysis only uses the call graph to handle context-sensitivity.
Example: Worst-Case Occupancy

function B()
sres 3
\text{call C()}
sens 3
\text{call C()}
sens 3
\text{sfree 3}

Stack cache full
Example: Worst-Case Occupancy

function B()
sres 3
call C()        4
call C()          disp(C) = 2
sens 3
sens 3
call C()       2
sfree 3

sres 3
call C()        4
call C()          disp(C) = 2
sens 3
sens 3
call C()       2
sfree 3
Example: Worst-Case Occupancy

```plaintext
function B()
sres 3
call C()
sens 3
call C()
sens 3
sfree 3
2
Fill 1 blocks
3
```

18/27
Example: Worst-Case Occupancy

function B()
sres 3 ............... 4
call C() 
sens 3 ................ 3
call C() 
sens 3 
sfree 3
Example: Spill Cost Graph

(a) Call graph

(b) Spill cost graph (construction)
Example: Spill Cost Graph

(a) Call graph

(b) Spill cost graph (construction)
Example: Spill Cost Graph

(a) Call graph

(b) Spill cost graph (construction)
Example: Spill Cost Graph

(a) Call graph

(b) Spill cost graph (construction)
Example: Spill Cost Graph

(a) Call graph

(b) Spill cost graph (construction)
Experimental Setup

- MiBench benchmark suite
- LLVM compiler 3.3 for the Patmos processor
- Stack cache configurations: 256B, 512B, 1kB
- Compile benchmarks and perform stack cache analysis
  - Context-insensitive Ensure Analysis
  - Fully context-sensitive Reserve Analysis
Experiments: Ensure Analysis

- adpcm
- crc-32
- fft
- bf
- rijndael
- sha
- say
- search
- dijkstra
- patricia
- basicmath
- bitcnts
- qsort
- susan
- ansi2knr
- cjpeg
- djpeg
- jpegtran
- jpgcom
- lame
- tiff2bw
- tiff2rgba
- tiffmedian

Number of ensure instructions

- sens instructions (total)
- sens filling (1k)
- sens filling (512b)
- sens filling (256b)
Experiments: Reserve Analysis

- Adpcm
- Crc-32
- Fft
- Bf
- Rijndael
- Sha
- Say
- Search
- Dijkstra
- Patricia
- Basicmath
- Bitcnts
- Qsort
- Susan
- Ansi2knr
- Cjpeg
- Djpeg
- Jpegtran
- Jpgcom
- Lane
- Tiff2bw
- Tiff2rgba
- Tiffdither
- Tiffmedian
Conclusion

• Novel cache design dedicated to stack data
• Analyzable caching strategy
• Does not require knowledge of addresses
• Does not require analysis of individual accesses
• Simple analysis
  • Compute displacement on call graph
  • Perform function-local data-flow analysis
  • Compute Spill Cost Graph on call graph
Why use a Stack Cache?

Normalized data transfer volume between the Patmos CPU and its data caches.

DC ... data cache  SC ... stack cache
Handling Recursion

Unbounded longest paths/tails
- Requires (user-supplied) bounds of recursion depth
- Modeled using Integer Linear Programming (ILP)
  - Recursion bounded by (user-supplied) ILP constraints
  - Model nested calls as flow in/out of call graph nodes
- Longest tail
  - Duplicate call graph
  - One copy contains zero-weighted edges
  - One copy contains the original edge weights
  - Transition edges from one copy to the other
Example: Displacement with Recursion

Search the path with the longest tail starting at C()

(a) Original call graph

(b) Transformed call graph
Extensions

- Generalized placement of stack control instructions
- Well-formed programs
  - Recursive definition based on well-formed paths
  - Matching arguments at first reserve \( r \) and last free \( f \)
  - The sub-path \( r \leadsto f \) is well-formed
- Required changes
  - Computation of edge weights in weighted call graph (depth first search)
  - Minor modifications to function-local data-flow analysis
  - Representation of contexts within functions
- Pruning of Spill Cost Graph
  - Lossless elimination of contexts (no impact on spill cost)
  - Lossy merging of similar contexts (impact on spill cost)