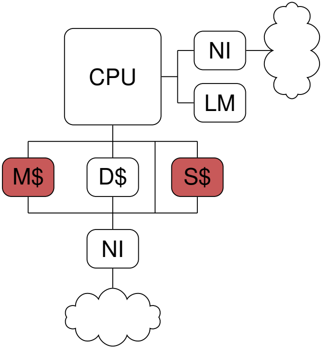
Time-predictable ~~(stack)~~ caches and their analysis

Florian Brandner
LTCI, CNRS, Télécom ParisTech, Université Paris-Saclay

The Patmos Multi-Core Platform

Time-predictable processor:

- Two non-standard caches
- Stack cache (S$):
  - Stack-related data only
    (often: content of spilled registers)
- Method cache (M$):
  - Instructions only
  - Variable-sized code blocks



A tile of the Patmos platform.

The Method Cache

The Method Cache

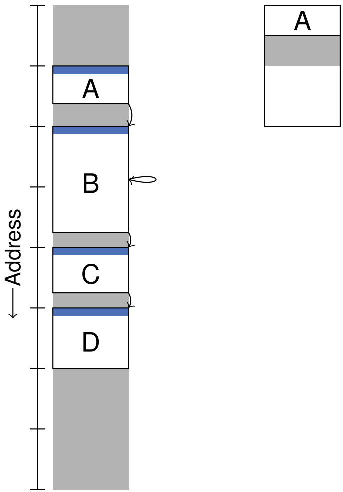
Predictable cache design for code:

- Simple ring buffer (FIFO replacement)[1]
- Caches variable-sized **code blocks**
- Cache misses at well-defined instructions only
  - `call` *x*: call function *x*
  - `br_cf` *x*: branch to code block *x*
- No cache misses for other instructions
  - `br` *x*: branch to instruction *x* within code block
  - No need to analyze `br`-style branches or any other instruction

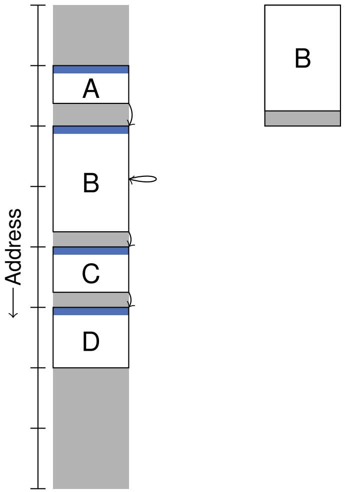
[1] LRU possible, but complicates hardware

TELECOM ParisTech

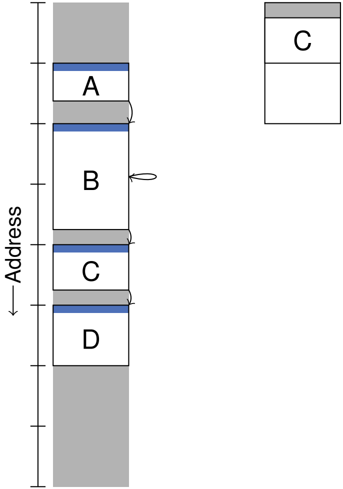
Example: Method Cache



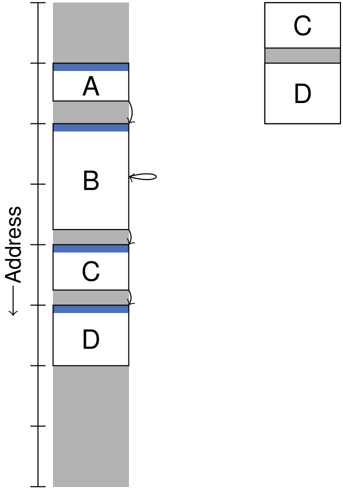
Example: Method Cache



Example: Method Cache



Example: Method Cache



Optimization and Analysis Problems

Method cache design relies heavily on software support:

- Compiler optimization
  - Split code of functions, forming code blocks
  - Combining the most profitable code parts into blocks
- WCET analysis
  - Determine the number of cache misses in the worst case
  - Find cache-conflict-free regions in the program
  - Somewhat related to persistence analysis

Optimization and Analysis Problems

Method cache design relies heavily on software support:

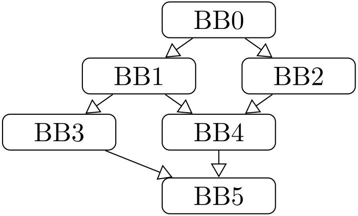
- Compiler optimization
  - Split code of functions, forming code blocks
  - Combining the most profitable code parts into blocks
- WCET analysis
  - Determine the number of cache misses in the worst case
  - Find cache-conflict-free regions in the program
  - Somewhat related to persistence analysis

Both problems are related, asking for a partitioning of the program's control-flow graph under size constraints.

Function Splitting

Split functions into code blocks:

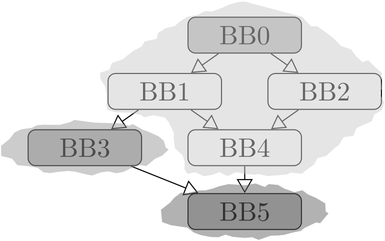
- Blocks have a single entry
- Code fits into the cache
- Observation:
  - Entry dominates other code
  - Loops either fit entirely or are split
- Elegant DFS-based algorithm



Function Splitting

Split functions into code blocks:

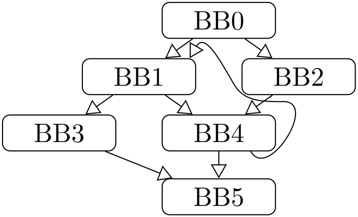
- Blocks have a single entry
- Code fits into the cache
- Observation:
  - Entry dominates other code
  - Loops either fit entirely or are split
- Elegant DFS-based algorithm



Function Splitting (2)

Works on all programs:

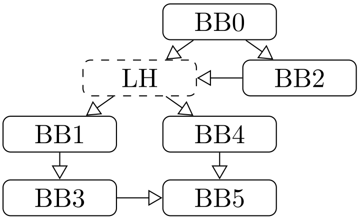
- Irreducible loops
  (loops with multiple entries)
- Computed branches
  (modeled as irreducible loops)



Function Splitting (2)

Works on all programs:

- Irreducible loops
  (loops with multiple entries)
- Computed branches
  (modeled as irreducible loops)



The 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

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

Logical stack

Stack cache*

*Cache configuration: 4 blocks

Example: Stack Cache

```
function A()        function B()        function C()
sres 2 ←            sres 2              sres 3
call B()            call C()            sfree 3
sens 2              sens 2
sfree 2             sfree 2
```

Logical stack

A	A							

Stack cache*

A	A		

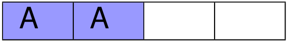
*Cache configuration: 4 blocks

Example: Stack Cache

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

Logical stack

Stack cache*



*Cache configuration: 4 blocks

Example: Stack Cache

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

Logical stack

A	A	B	B					

Stack cache*

A	A	B	B

*Cache configuration: 4 blocks

Example: Stack Cache

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

```
function B()
sres 2
call C() ←
sens 2
sfree 2
```

```
function C()
sres 3
sfree 3
```

Logical stack

A	A	B	B					

Stack cache*

A	A	B	B

*Cache configuration: 4 blocks

Example: Stack Cache

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

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

```
function C()
sres 3 ←
sfree 3
```

Logical stack

A	A	B	B	C	C	C		

Stack cache*

B	C	C	C

*Cache configuration: 4 blocks

Example: Stack Cache

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

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

```
function C()
sres 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 2           sres 3
(3) call B()         call C()         sfree 3
(4) sens 2           sens 2 ←
(5) sfree 2          sfree 2
```

Logical stack

A	A	B	B					

Stack cache*

B	B		

*Cache configuration: 4 blocks

Example: Stack Cache

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

```
function B()
sres 2
call C()
sens 2
sfree 2 ←
```

```
function C()
sres 3
sfree 3
```

Logical stack

A	A							

Stack cache*

*Cache configuration: 4 blocks

Example: Stack Cache

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

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 2            sres 3
(3) call B()          call C()          sfree 3
(4) sens 2            sens 2
(5) sfree 2 ←         sfree 2
```

Logical stack

Stack cache*

*Cache configuration: 4 blocks

Analysis and Optimization Problems

- Compiler optimization:
  - Placement of stack control instructions (simple for now)
  - Consider stack cache during register allocation
  - Optimize stack frame layout
- WCET analysis:
  - Determine maximum filling/spilling at `sens` and `sres`
  - Depends on minimum/maximum <u>occupancy</u> level

TELECOM ParisTech

Stack Cache Analysis

Based on the following observation:

- Functions free stack space before returning
  - Occupancy after a call may only reduce or remain the same
  - The occupancy after a call only depends on the displacement of the called function
  - **Displacement:**
    Number of cache blocks evicted from the stack cache (minimum/maximum).
- Displacement implies function-local occupancy bounds

This results in a nice problem decomposition!

Stack Cache Analysis (2)

1. Compute minimum/maximum displacement
   (shortest/longest path search on weighted call graph)
2. Compute function-local occupancy bounds
   (function-local data-flow analysis)
3. Compute context-insensitive filling bounds
   - Directly derived from the function-local minimum occupancy
4. Compute fully context-sensitive spilling bounds
   - Propagate occupancy through call graph
   - Either take occupancy of calling context . . .
   - . . . or the local maximum occupancy bound

TELECOM ParisTech

Example: Stack Cache Analysis

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

- Displacement of `B` (minimum = maximum):

$$min(|SC|, 2 + 3) = min(4, 2 + 3) = 4$$

- Minimum local occupancy after returning from `B`:

$$|SC| - 4 = 0$$

- Maximum filling at `sens`: 2

*Cache configuration: $|SC| = 4$ blocks

TELECOM ParisTech

Example: Stack Cache Analysis

`(1) function A()`	`function B()`	`function C()`
`(2)` **`sres 2`**	**`sres 2`**	**`sres 3`**
`(3) call B()`	<u>`call C()`</u>	`sfree 3`
`(4) sens 2`	`sens 2`	
`(5) sfree 2`	`sfree 2`	

- Displacement of `C` (minimum = maximum):

$$min(|SC|, 3) = min(4, 3) = 3$$

- Local maximum occupancy after returning from `C`:

$$|SC| - 3 = 1$$

- Maximum occupancy with context: $min(2 + 2, 1) = 1$

*Cache configuration: $|SC| = 4$ blocks

Conclusion

Patmos platform offers two non-standard caches:

- Stack cache
  - Efficient and easy to analyze
  - Several extensions available
  - Weakness: task preemption
- Method cache
  - Flexible and fully compiler-controlled
  - Less studies with many open questions
  - Several extensions on the way
  - Weakness: task preemption, compiler support