Lazy Spilling for a Time-Predictable Stack Cache: Implementation and Analysis

Sahar Abbaspour, Alexander Jordan
Embedded Systems Engineering Sect.
Technical University of Denmark

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

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

Real-Time Systems

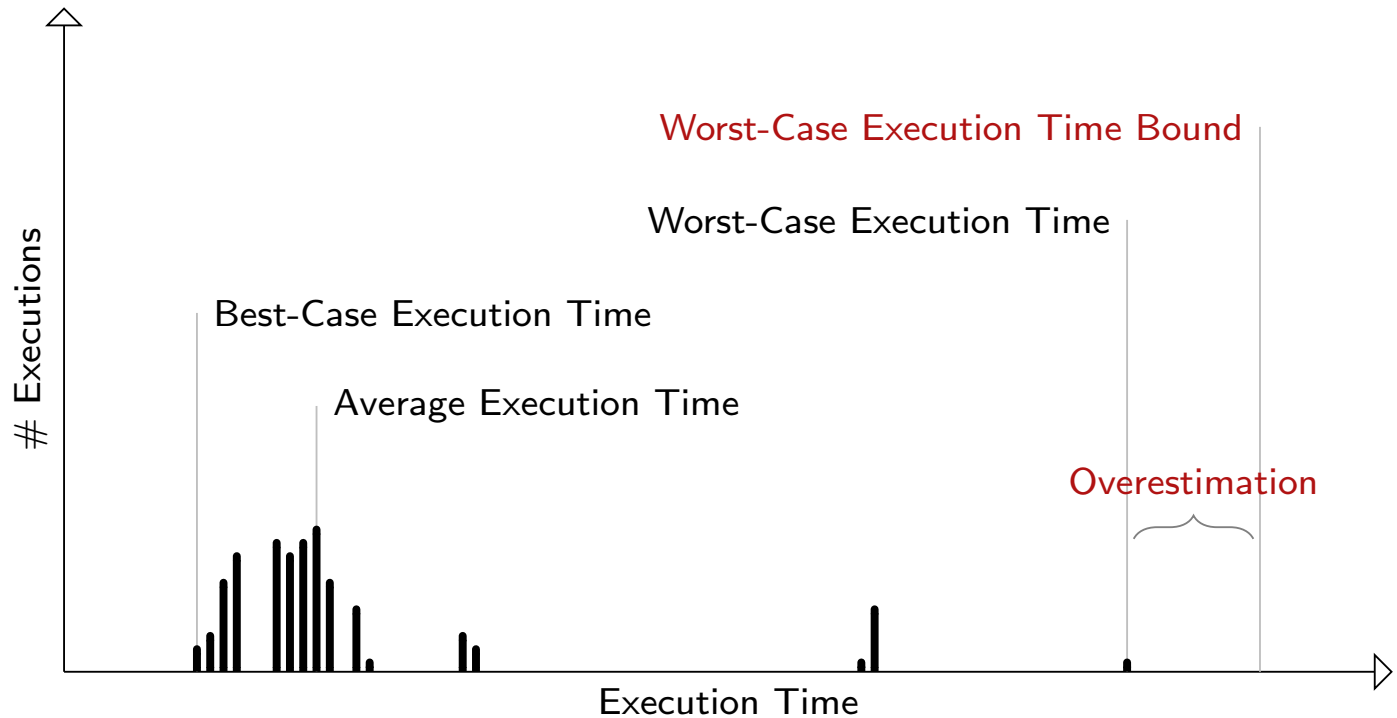
Strict timing **guarantees**

- 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

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**
  - Processor pipeline
  - Branch predictors
  - Data and instruction caches
  - Main memory

Example: Miss/Hit Classification

Initial cache state*

0x100	0x200
0x101	0x103

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

Example: Miss/Hit Classification

Initial cache state*

0x100	0x200
0x101	0x103

`lw [0x100]`

0x100	0x200
0x101	0x103

Classified as hit

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

Example: Miss/Hit Classification

Initial cache state*

0x100	0x200
0x101	0x103

`lw [0x100]`

0x100	0x200
0x101	0x103

Classified as hit

`lw [0x105]`

0x100	0x200
0x105	0x101

Classified as miss

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

Example: Miss/Hit Classification

Initial cache state*

0x100	0x200
0x101	0x103

`lw [0x100]`

0x100	0x200
0x101	0x103

Classified as hit

`lw [0x105]`

0x100	0x200
0x105	0x101

Classified as miss

`lw [??]`

??	??
??	??

Classification unclear

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

Example: Miss/Hit Classification

Initial cache state*

0x100	0x200
0x101	0x103

`lw [0x100]`

0x100	0x200
0x101	0x103

Classified as hit

`lw [0x105]`

0x100	0x200
0x105	0x101

Classified as miss

`lw [??]`

??	??
??	??

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

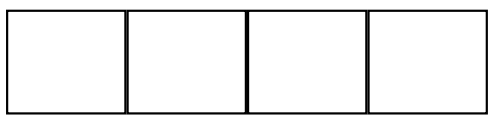
- Simple ring buffer (*FIFO replacement*)
- All stack accesses are guaranteed hits (no need to analyze them)
- Dedicated stack control instructions (need to be analyzed)
  - `sres` x: reserve x blocks on the stack
  - `sfree` x: free x blocks on the stack
  - `sens` x: ensure that at least x blocks are cached
- Intuitively: a cache window following the stack top
  - Implemented as two pointers
  - `MT`: Memory-Top
  - `ST`: Stack-Top

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*

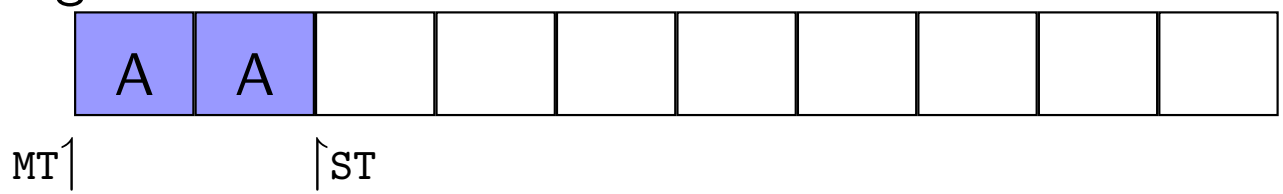


*Cache configuration: 4 blocks

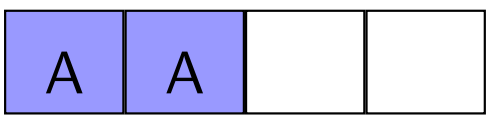
Example: Stack Cache

```
(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



Stack cache*

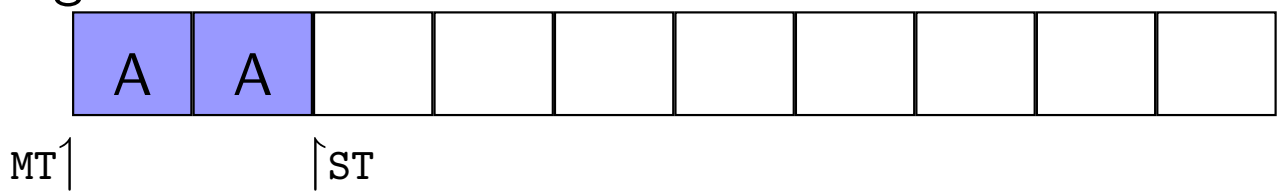


*Cache configuration: 4 blocks

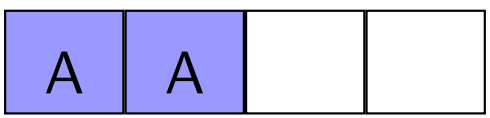
Example: Stack Cache

```
(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*

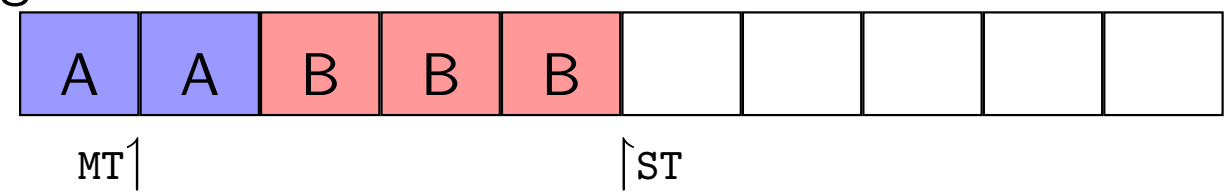


*Cache configuration: 4 blocks

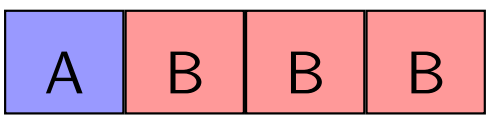
Example: Stack Cache

```
(1) function A()      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*



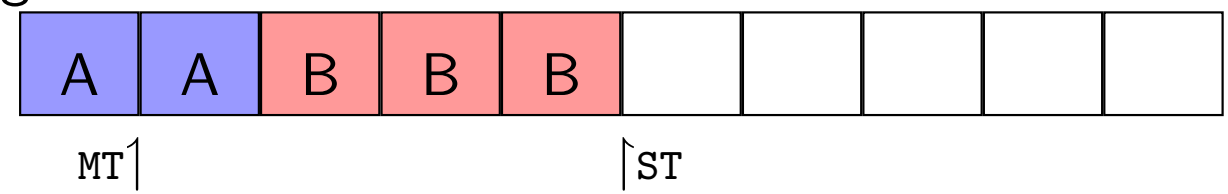
spill 1 block

*Cache configuration: 4 blocks

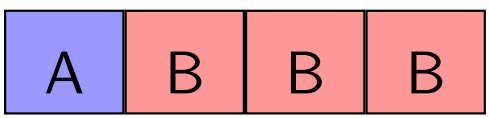
Example: Stack Cache

```
(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*

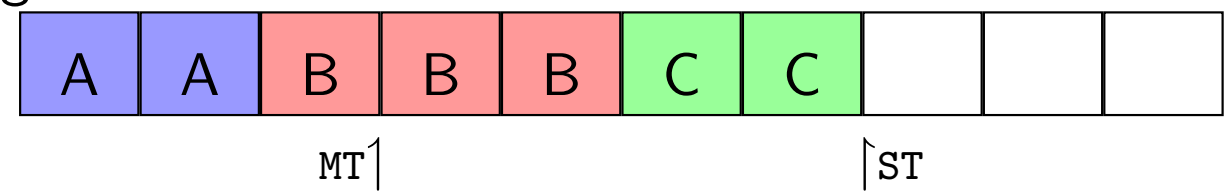


*Cache configuration: 4 blocks

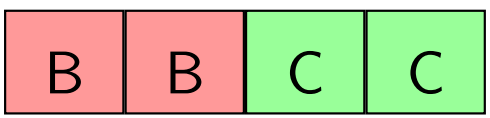
Example: Stack Cache

```
(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



Stack cache*



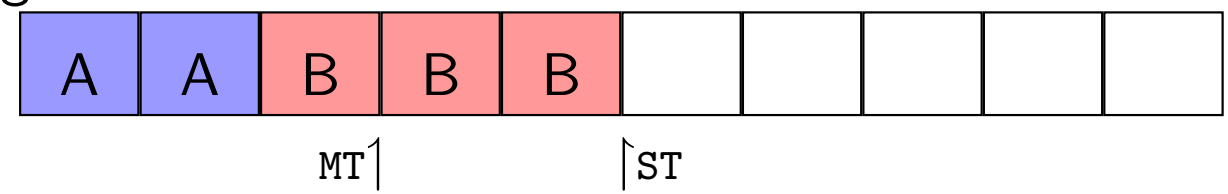
spill 2 blocks

*Cache configuration: 4 blocks

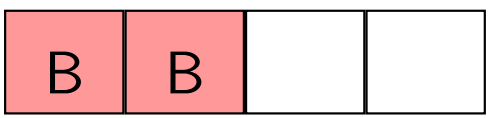
Example: Stack Cache

```
(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*

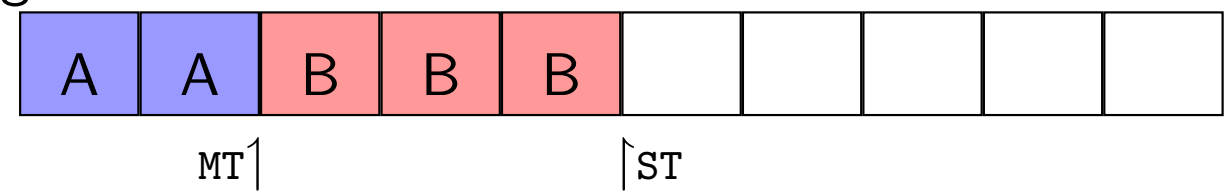


*Cache configuration: 4 blocks

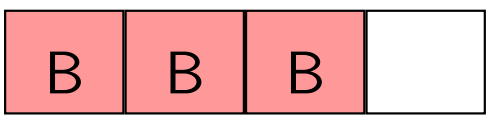
Example: Stack Cache

```
(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



Stack cache*



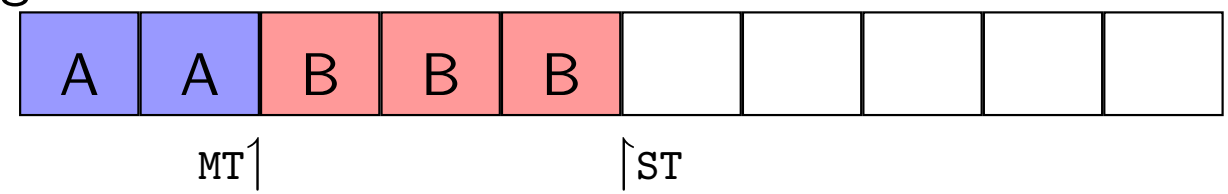
fill 1 block

*Cache configuration: 4 blocks

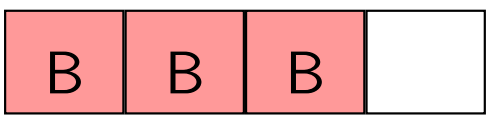
Example: Stack Cache

```
(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*

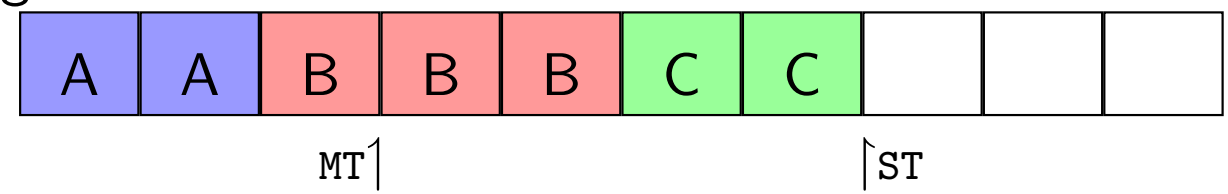


*Cache configuration: 4 blocks

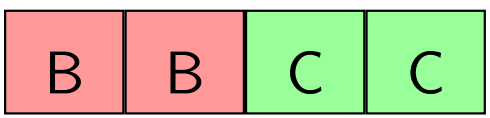
Example: Stack Cache

```
(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



Stack cache*



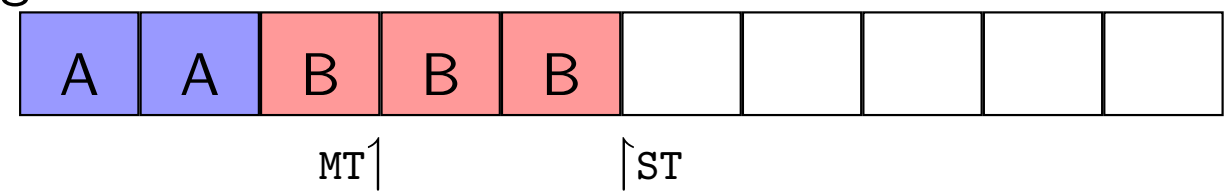
spill 1 block

*Cache configuration: 4 blocks

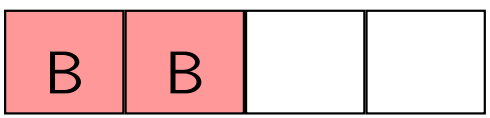
Example: Stack Cache

```
(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*

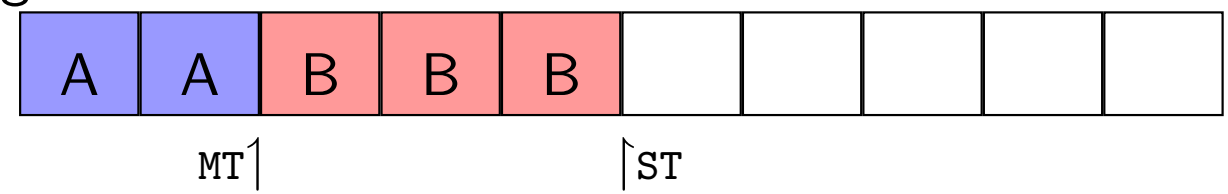


*Cache configuration: 4 blocks

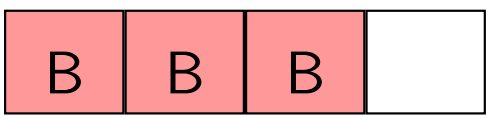
Example: Stack Cache

```
(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



Stack cache*



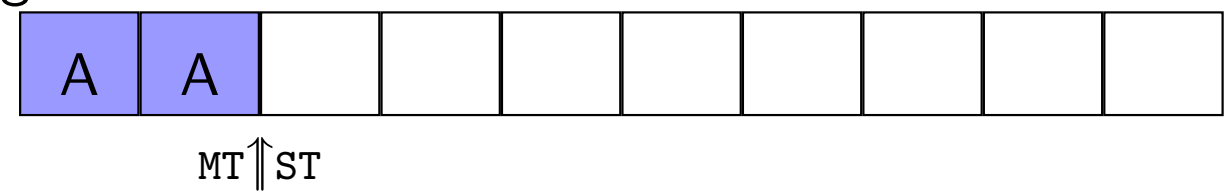
fill 1 block

*Cache configuration: 4 blocks

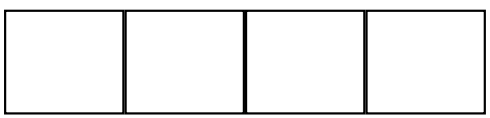
Example: Stack Cache

```
(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*

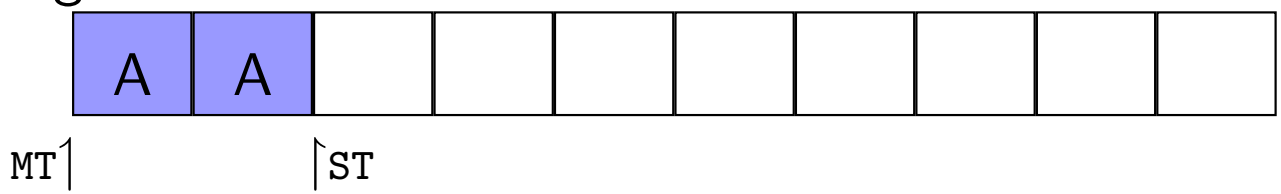


*Cache configuration: 4 blocks

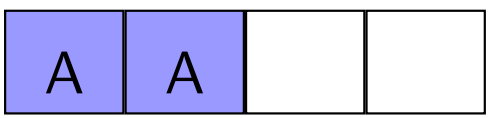
Example: Stack Cache

```
(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



Stack cache*



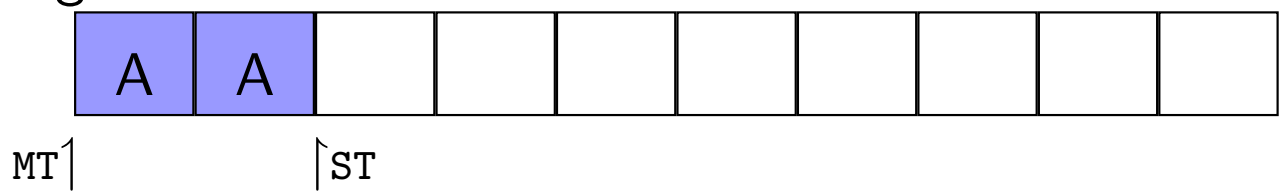
fill 2 blocks

*Cache configuration: 4 blocks

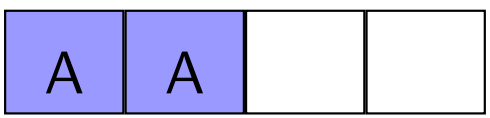
Example: Stack Cache

```
(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*

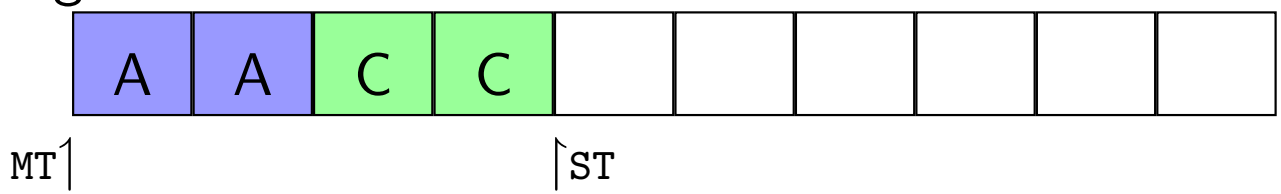


*Cache configuration: 4 blocks

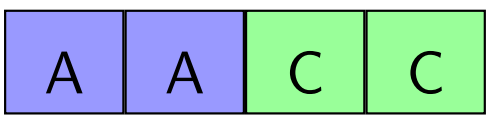
Example: Stack Cache

```
(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



Stack cache*

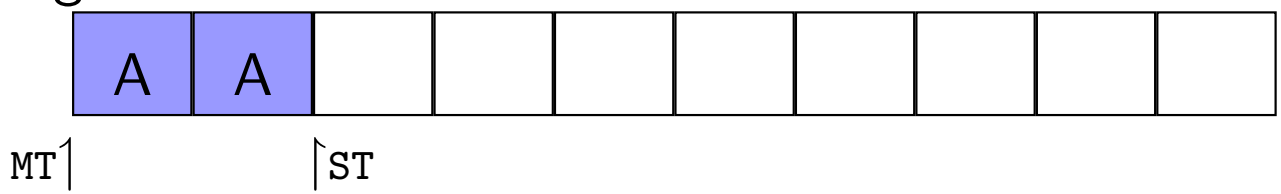


*Cache configuration: 4 blocks

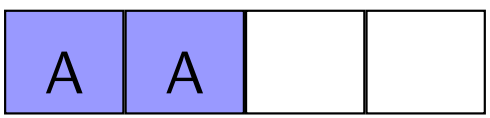
Example: Stack Cache

```
(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*

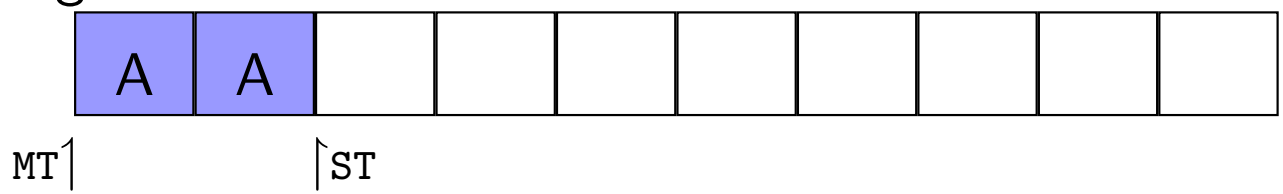


*Cache configuration: 4 blocks

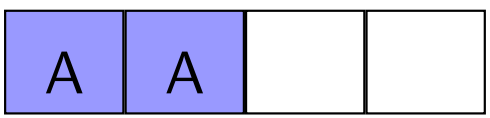
Example: Stack Cache

```
(1) function A()      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

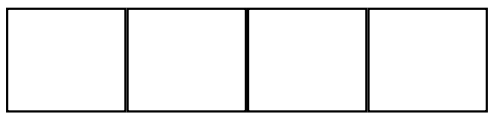
Example: Stack Cache

```
(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

MT⇑ST

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, i.e., $\texttt{MT} - \texttt{ST}$.

Displacement
Number of cache blocks spilled to main memory at a function call, i.e., $\texttt{MT}_{\texttt{before}} - \texttt{MT}_{\texttt{after}}$.

Terminology

Occupancy

Number of cache blocks utilized at a given program point, i.e., $\texttt{MT} - \texttt{ST}$.

Displacement

Number of cache blocks spilled to main memory at a function call, i.e., $\texttt{MT}_{\texttt{before}} - \texttt{MT}_{\texttt{after}}$.

Observations

Concrete values of ST or MT are not relevant (only differences).

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()`

$A()_3 \rightarrow B()_3 \rightarrow C()$: 4 $\rightarrow$ 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
(5) call C()            call C()
(6) sens 2              sens 3
(7) sfree 2             sfree 3
```

Occupancy at `C()`

$A()_3 \rightarrow B()_3 \rightarrow C()$: 4 $\rightarrow$ spill 2 blocks

$A()_3 \rightarrow B()_5 \rightarrow C()$: 3 $\rightarrow$ spill 1 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
(5) call C()            call C()
(6) sens 2              sens 3
(7) sfree 2             sfree 3
```

Occupancy at `C()`

$A()_3 \rightarrow B()_3 \rightarrow C()$: 4 $\rightarrow$ spill 2 blocks

$A()_3 \rightarrow B()_5 \rightarrow C()$: 3 $\rightarrow$ spill 1 blocks

$A()_5 \rightarrow C()$: 2 $\rightarrow$ no spilling

Stack Cache Analysis

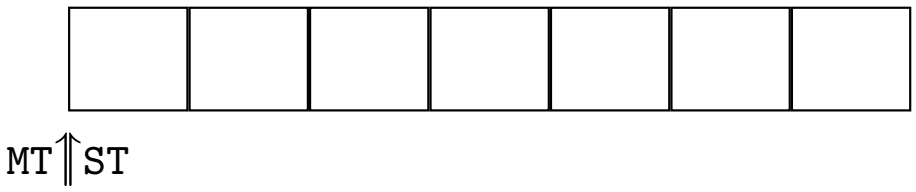
Bound occupancy at stack cache instructions

1. Pre-compute the *minimum/maximum* displacement at calls
   (shortest/longest path search on weighted call graph)
2. Perform a function-local data-flow analyses
   - Propagate the minimum/maximum occupancy
   - Adjust occupancy at `sens`-instructions
   - Adjust occupancy at calls using max./min. displacement
3. Bound worst-case filling using the minimum occupancy
   (context insensitive)
4. Bound worst-case spilling using the maximum occupancy
   (fully context-sensitive, on call graph only!)

Lazy Spilling

Motivating Example

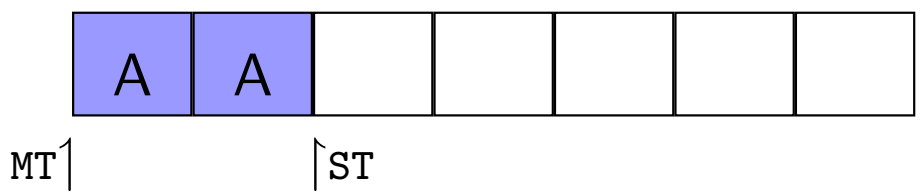
```
function A()
sres 2
sws [1] = ...     // store loop-invariant stack data
loop:
  lws ...  = [1]  // load loop-invariant stack data
  ...
  call B          // displaces entire stack cache
  sens 2          // reload local stack frame
  ...
  bt loop        // jump to beginning of loop
// exit function
sfree 2
```



* Cache configuration: 4 blocks

Motivating Example

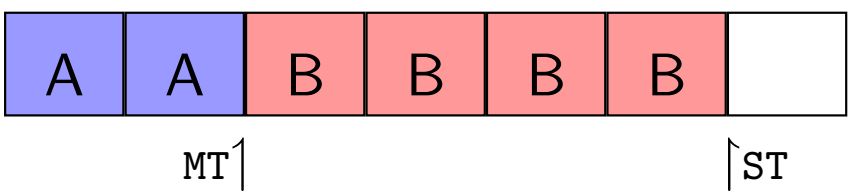
```
function A()
sres 2 ←
sws [1] = ...      // store loop-invariant stack data
loop:
  lws ...  = [1]   // load loop-invariant stack data
  ...
  call B           // displaces entire stack cache
  sens 2           // reload local stack frame
  ...
  bt loop          // jump to beginning of loop
// exit function
sfree 2
```



*Cache configuration: 4 blocks

Motivating Example

```
function A()
sres 2
sws [1] = ...      // store loop-invariant stack data
loop:
  lws ...  = [1]   // load loop-invariant stack data
  ...
  call B ←         // displaces entire stack cache
  sens 2           // reload local stack frame
  ...
  bt loop          // jump to beginning of loop
// exit function
sfree 2
```

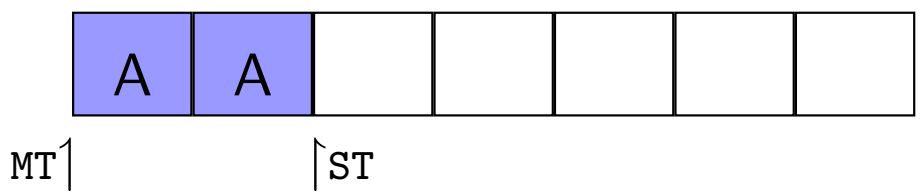


spilling of 2 blocks

*Cache configuration: 4 blocks

Motivating Example

```
function A()
sres 2
sws [1] = ...      // store loop-invariant stack data
loop:
  lws ...  = [1]   // load loop-invariant stack data
  ...
  call B           // displaces entire stack cache
  sens 2 ←         // reload local stack frame
  ...
  bt loop          // jump to beginning of loop
// exit function
sfree 2
```

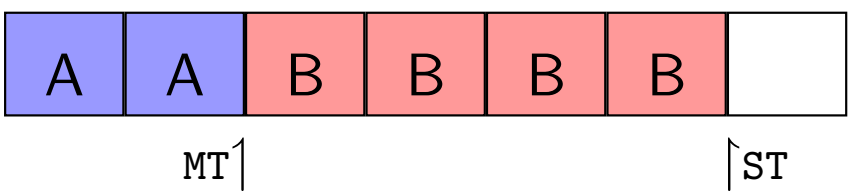


filling of 2 blocks

* Cache configuration: 4 blocks

Motivating Example

```
function A()
sres 2
sws [1] = ...      // store loop-invariant stack data
loop:
  lws ...  = [1]   // load loop-invariant stack data
  ...
  call B ←         // displaces entire stack cache
  sens 2           // reload local stack frame
  ...
  bt loop          // jump to beginning of loop
// exit function
sfree 2
```

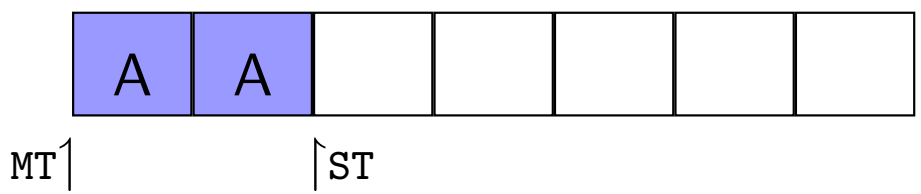


useless spilling of 2 unmodified blocks!

*Cache configuration: 4 blocks

Motivating Example

```
function A()
sres 2
sws [1] = ...      // store loop-invariant stack data
loop:
  lws ...  = [1]   // load loop-invariant stack data
  ...
  call B           // displaces entire stack cache
  sens 2 ←         // reload local stack frame
  ...
  bt loop          // jump to beginning of loop
// exit function
sfree 2
```

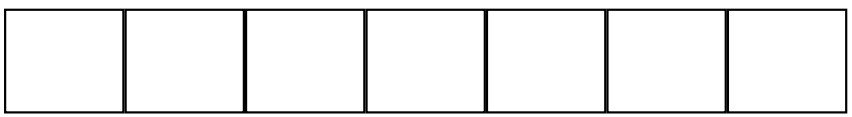


filling of 2 blocks

* Cache configuration: 4 blocks

Motivating Example

```
function A()
sres 2
sws [1] = ...      // store loop-invariant stack data
loop:
  lws ...  = [1]   // load loop-invariant stack data
  ...
  call B           // displaces entire stack cache
  sens 2           // reload local stack frame
  ...
  bt loop          // jump to beginning of loop
// exit function
sfree 2 ←
```



* Cache configuration: 4 blocks

Lazy Spilling

Basic idea:

- Avoid redundant spilling of coherent data

Lazy Spilling

Basic idea:

- Avoid redundant spilling of coherent data
- Keeping track of coherent data
    - Cached stack slots whose value is the *same* in main memory

Lazy Spilling

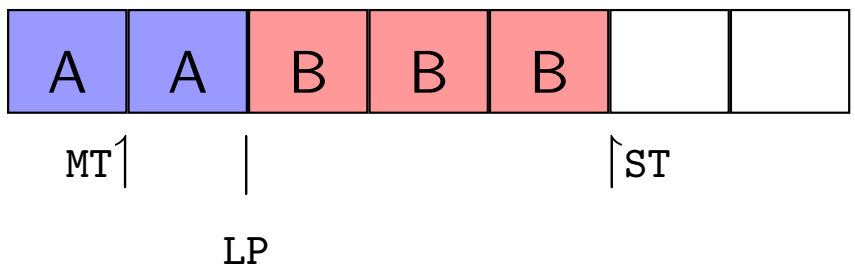
Basic idea:

- Avoid redundant spilling of coherent data
- Keeping track of coherent data
  - Cached stack slots whose value is the *same* in main memory
  - Introduce a *Lazy Pointer* (`LP`)

Lazy Spilling

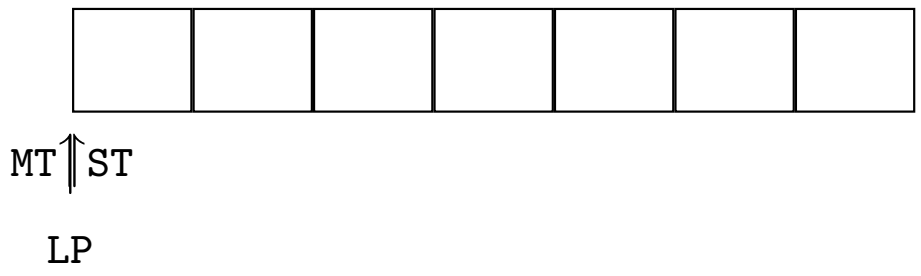
Basic idea:

- Avoid redundant spilling of coherent data
- Keeping track of coherent data
  - Cached stack slots whose value is the *same* in main memory
  - Introduce a *Lazy Pointer* (LP)
  - Data between MT and LP is coherent ($\texttt{ST} \leq \texttt{LP} \leq \texttt{MT}$)
  - Illustration:



Motivating Example Revisited

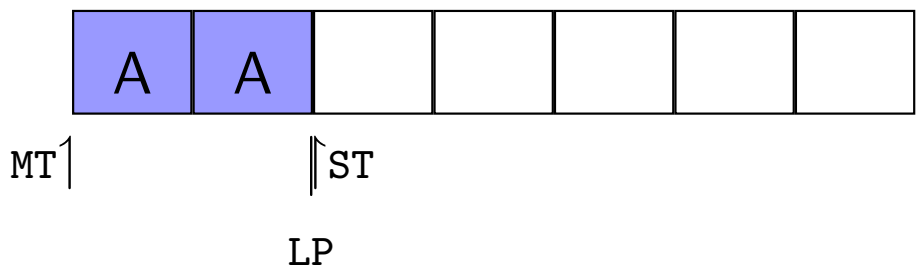
```
function A()
sres 2
sws [0] = ...      // store loop-invariant stack data
loop:
  lws ...  = [0]  // load loop-invariant stack data
  ...
  call B          // displaces entire stack cache
  sens 2          // reload local stack frame
  ...
  bt loop        // jump to beginning of loop
// exit function
sfree 2
```



*Cache configuration: 4 blocks

Motivating Example Revisited

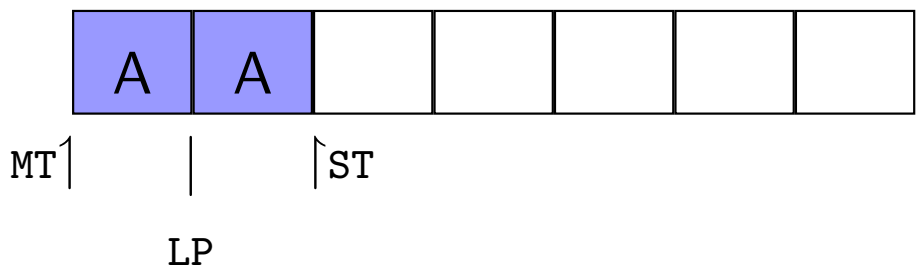
```
function A()
sres 2 ←
sws [0] = ...      // store loop-invariant stack data
loop:
  lws ...  = [0]  // load loop-invariant stack data
  ...
  call B          // displaces entire stack cache
  sens 2          // reload local stack frame
  ...
  bt loop        // jump to beginning of loop
// exit function
sfree 2
```



*Cache configuration: 4 blocks

Motivating Example Revisited

```
function A()
sres 2
sws [0] = ... ←   // store loop-invariant stack data
loop:
  lws ...  = [0]  // load loop-invariant stack data
  ...
  call B          // displaces entire stack cache
  sens 2          // reload local stack frame
  ...
  bt loop         // jump to beginning of loop
// exit function
sfree 2
```

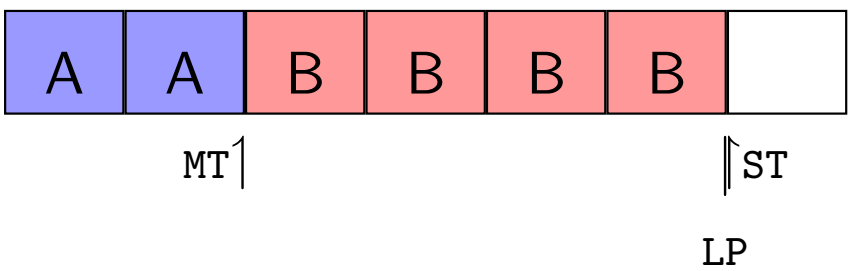


modifying 1 block

*Cache configuration: 4 blocks

Motivating Example Revisited

```
function A()
sres 2
sws [0] = ...      // store loop-invariant stack data
loop:
  lws ...  = [0]  // load loop-invariant stack data
  ...
  call B ←       // displaces entire stack cache
  sens 2         // reload local stack frame
  ...
  bt loop        // jump to beginning of loop
// exit function
sfree 2
```

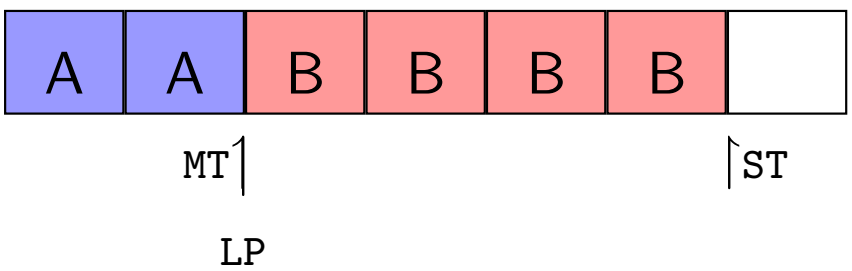


spilling of 1 block

*Cache configuration: 4 blocks

Motivating Example Revisited

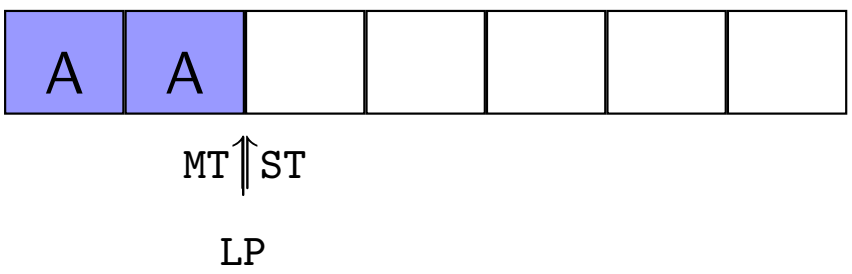
```
function A()
sres 2
sws [0] = ...      // store loop-invariant stack data
loop:
  lws ...  = [0]  // load loop-invariant stack data
  ...
  call B ←       // displaces entire stack cache
  sens 2         // reload local stack frame
  ...
  bt loop        // jump to beginning of loop
// exit function
sfree 2
```



*Cache configuration: 4 blocks

Motivating Example Revisited

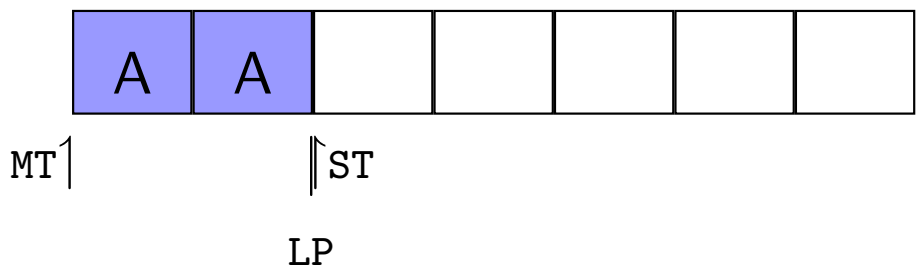
```
function A()
sres 2
sws [0] = ...      // store loop-invariant stack data
loop:
  lws ...  = [0]  // load loop-invariant stack data
  ...
  call B ←       // displaces entire stack cache
  sens 2         // reload local stack frame
  ...
  bt loop        // jump to beginning of loop
// exit function
sfree 2
```



*Cache configuration: 4 blocks

Motivating Example Revisited

```
function A()
sres 2
sws [0] = ...      // store loop-invariant stack data
loop:
  lws ...  = [0]  // load loop-invariant stack data
  ...
  call B          // displaces entire stack cache
  sens 2 ←        // reload local stack frame
  ...
  bt loop         // jump to beginning of loop
// exit function
sfree 2
```

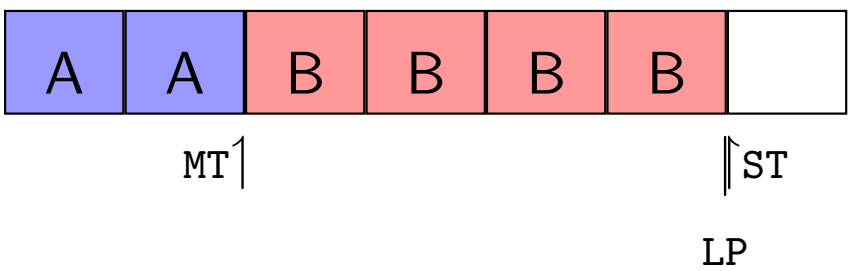


filling of 2 blocks

*Cache configuration: 4 blocks

Motivating Example Revisited

```
function A()
sres 2
sws [0] = ...      // store loop-invariant stack data
loop:
  lws ...  = [0]  // load loop-invariant stack data
  ...
  call B ←       // displaces entire stack cache
  sens 2         // reload local stack frame
  ...
  bt loop        // jump to beginning of loop
// exit function
sfree 2
```

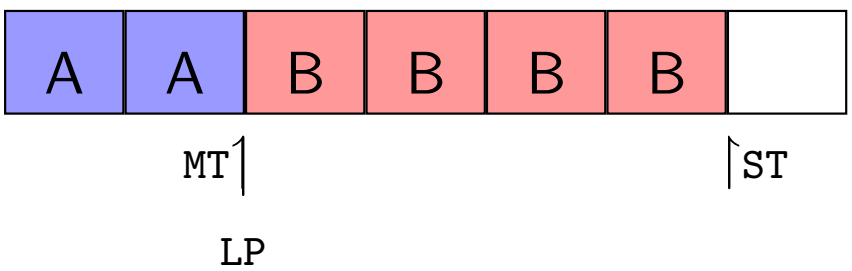


no spilling!

*Cache configuration: 4 blocks

Motivating Example Revisited

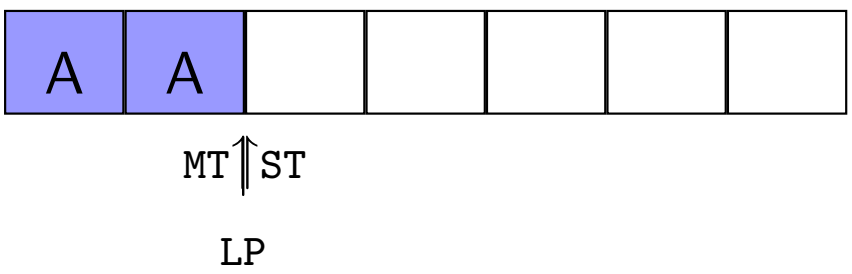
```
function A()
sres 2
sws [0] = ...      // store loop-invariant stack data
loop:
  lws ...  = [0]  // load loop-invariant stack data
  ...
  call B ←       // displaces entire stack cache
  sens 2         // reload local stack frame
  ...
  bt loop        // jump to beginning of loop
// exit function
sfree 2
```



*Cache configuration: 4 blocks

Motivating Example Revisited

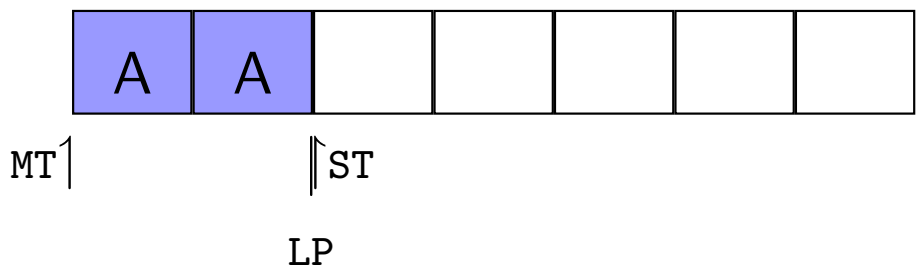
```
function A()
sres 2
sws [0] = ...      // store loop-invariant stack data
loop:
  lws ...  = [0]  // load loop-invariant stack data
  ...
  call B ←       // displaces entire stack cache
  sens 2         // reload local stack frame
  ...
  bt loop        // jump to beginning of loop
// exit function
sfree 2
```



*Cache configuration: 4 blocks

Motivating Example Revisited

```
function A()
sres 2
sws [0] = ...      // store loop-invariant stack data
loop:
  lws ...  = [0]  // load loop-invariant stack data
  ...
  call B          // displaces entire stack cache
  sens 2 ←        // reload local stack frame
  ...
  bt loop         // jump to beginning of loop
// exit function
sfree 2
```

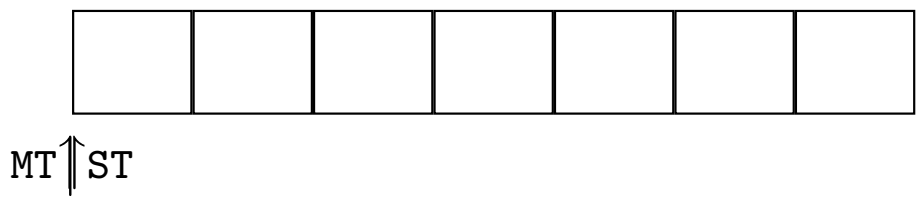


filling of 2 blocks

*Cache configuration: 4 blocks

Motivating Example Revisited

```
function A()
sres 2
sws [0] = ...      // store loop-invariant stack data
loop:
  lws ...  = [0]  // load loop-invariant stack data
  ...
  call B          // displaces entire stack cache
  sens 2          // reload local stack frame
  ...
  bt loop        // jump to beginning of loop
// exit function
sfree 2 ←
```



*Cache configuration: 4 blocks

Hardware Impact

Minor changes only:

`sens`	• No changes at all
`sres`	• No changes w.r.t. `MT` and `ST` • Replace `MT` by `LP` for spilling • Ensure that `ST` $\leq$ `LP` $\leq$ `MT`
`sfree`	• Ensure that `ST` $\leq$ `LP` $\leq$ `MT`
`s[whb]s`	• Ensure that `LP` is above effective address

Revisiting Terminology

Occupancy (unchanged)

Displacement (unchanged)

Effective Occupancy (new)
Number of dirty cache blocks at a given program point, i.e., $\texttt{LP} - \texttt{ST}$.

Revisiting Terminology

Occupancy (unchanged)

Displacement (unchanged)

Effective Occupancy (new)
Number of dirty cache blocks at a given program point, i.e., $\texttt{LP} - \texttt{ST}$.

Observations

Concrete values of `LP`, `ST`, or `MT` are not relevant (only differences).

The `LP` with regard to a function may only go *down* when calling other functions, i.e., the effective occupancy decreases.

WCET Analysis

Still analyzable:

- No change to min./max. displacement

WCET Analysis

Still analyzable:

- No change to min./max. displacement
- No change to ensure analysis

WCET Analysis

Still analyzable:

- No change to min./max. displacement
- No change to ensure analysis
- Reserve analysis:
  - Needs to account for stores in local data-flow analysis
  - Can mostly ignore `sens` instructions
  - Rest remains mostly unchanged

WCET Analysis

Still analyzable:

- No change to min./max. displacement
- No change to ensure analysis
- Reserve analysis:
  - Needs to account for stores in local data-flow analysis
  - Can mostly ignore `sens` instructions
  - Rest remains mostly unchanged
- DONE! :-)

Experimental Setup

- MiBench benchmark suite
- LLVM compiler 3.3 for the Patmos processor
- Stack cache configurations: 128B and 256B
- Compile benchmarks and perform stack cache analysis
  - Context-insensitive Ensure Analysis
  - Fully context-sensitive Reserve Analysis
- Execute benchmarks
  - Compare analysis against data from traces (not the worst-case)
  - Compare cache efficiency (not cache-miss rate)

Experiments: Cache Efficiency

- Reduction in number of blocks spilled (Spill)
- Efficiency: $\frac{\#RD+\#WR}{\#Stalls}$ for stack cache (SC/LP-SC) and data cache (DC)

	SC_{128}			LP_{128}		SC_{256}			LP_{256}		DC
Benchmark	SC	DC	Spill	LP-SC	DC	SC	DC	Spill	LP-SC	DC	
basicmath-tiny	2.3	1.1	0.17	4.0	1.1	26.4	1.1	0.53	34.0	1.1	1.1
bitcnts	4.6	191.6	0.00	12.2	191.6	17054.7	193.7	0.71	19201.4	193.7	1.2
cjpeg-small	116.9	1.0	0.51	148.4	1.0	3470.7	1.0	0.09	6154.4	1.0	1.1
crc-32	9.0	0.9	0.03	21.3	0.9	814.9	0.9	1.00	814.9	0.9	0.9
csusan-small	11.3	2.2	0.16	18.6	2.2	1218.8	2.3	0.72	1430.0	2.3	1.5
dbf	477.4	1.0	0.47	623.0	1.0	–	1.0	–	–	1.0	1.0
dijkstra-small	19.5	1.4	0.20	32.8	1.4	335.2	1.4	0.54	433.7	1.4	1.4
djpeg-small	9.0	0.8	0.34	13.5	0.8	293.4	0.8	0.66	361.5	0.8	0.8
drijndael	15.8	0.9	0.20	28.7	0.9	185620.0	0.9	1.00	185620.0	0.9	0.9
ebf	172.5	1.0	0.44	224.6	1.0	–	1.0	–	–	1.0	1.0
erijndael	32.6	0.9	0.57	43.3	0.9	258340.0	0.9	1.00	258340.0	0.9	0.9
esusan-small	15.9	3.4	0.25	25.3	3.4	70.7	3.6	0.02	139.5	3.6	1.5
fft-tiny	3.1	1.1	0.08	5.8	1.1	85.0	1.1	0.56	103.4	1.1	1.1
ifft-tiny	3.1	1.2	0.08	5.9	1.2	83.1	1.1	0.56	101.0	1.1	1.1
patricia	2.5	1.0	0.27	4.2	1.0	26.4	1.0	0.55	31.9	1.0	1.0
qsort-small	3.1	1.0	0.62	3.7	1.0	7.8	1.0	0.76	8.6	1.0	1.0
rsynth-tiny	16.0	1.9	0.08	29.9	1.9	1096.1	1.9	0.48	1539.8	1.9	1.3
search-large	2.9	0.8	0.48	3.9	0.8	26.3	0.8	0.00	52.5	0.8	0.9
search-small	2.9	0.8	0.49	3.7	0.8	28.1	0.8	0.02	54.8	0.8	0.9
sha	8.3	1.6	0.20	14.1	1.6	668.7	1.6	0.91	700.6	1.6	1.6
ssusan-small	29.2	17.1	0.20	43.9	17.1	4313.5	17.1	0.80	4678.0	17.1	3.3

Experiments: Analysis Precision

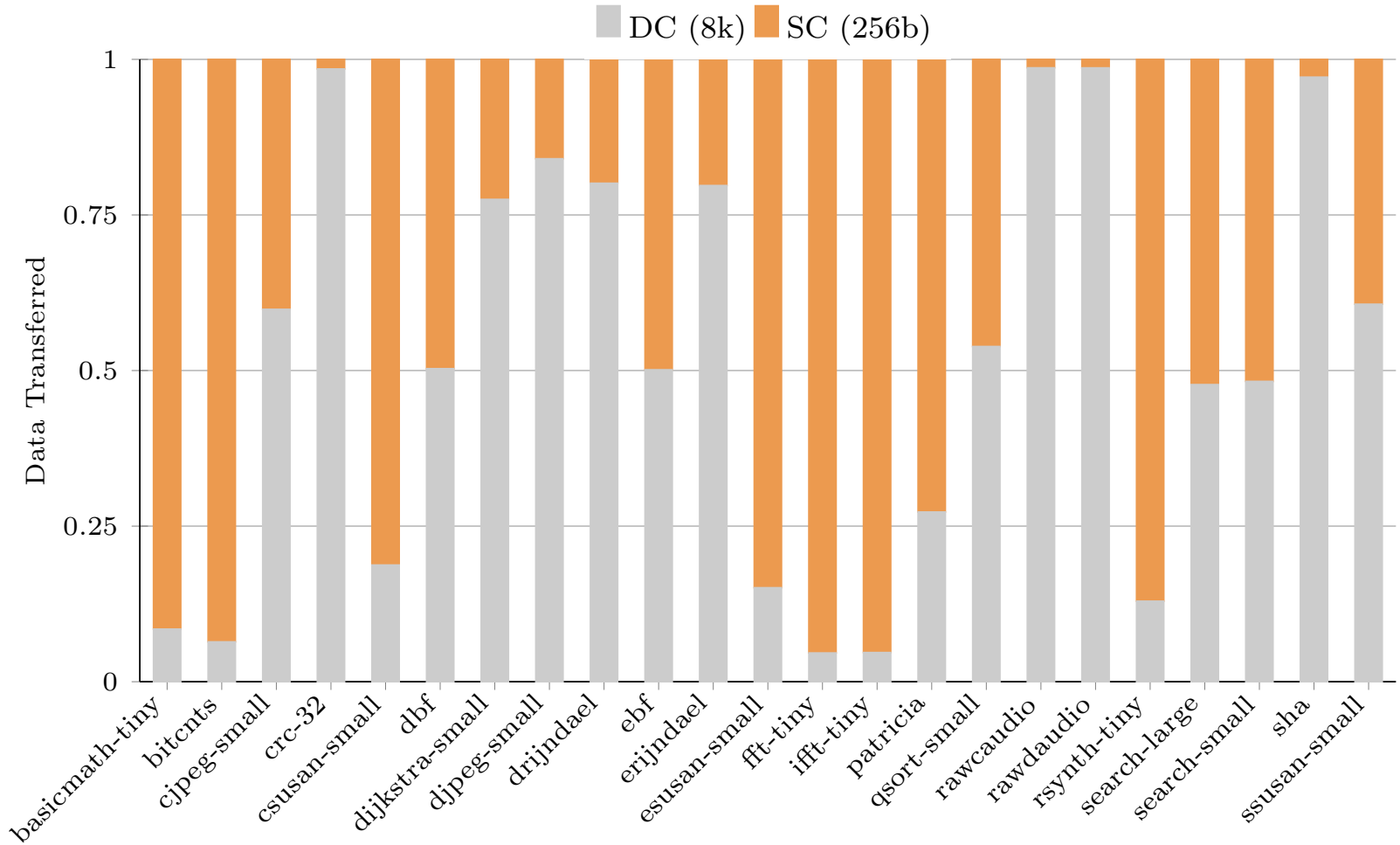
- Number of blocks spilled in trace (Dynamic)
- Predicted worst-case number of blocks spilled (Static)

	SC_{128} Max-Spilling-Δ			LP-SC_{128} Max-Spilling-Δ		
Benchmark	Static	Dynamic	Gap	Static	Dynamic	Gap
basicmath-tiny	68,128	32,040	2.13×	10,052	8,080	1.24×
bitcnts	892	684	1.30×	768	320	2.40×
crc-32	844	652	1.29×	684	372	1.84×
csusan	5,404	2,592	2.08×	2,420	1,196	2.02×
dbf	684	456	1.50×	564	324	1.74×
dijkstra-small	10,220	5,796	1.76×	6,676	2,608	2.56×
drijndael	1,172	664	1.77×	1,024	488	2.10×
ebf	684	456	1.50×	564	324	1.74×
erijndael	880	400	2.20×	752	292	2.58×
esusan	4,724	1,888	2.50×	2,256	1,024	2.20×
fft-tiny	32,484	9,476	3.43×	5,804	3,712	1.56×
ifft-tiny	32,224	9,256	3.48×	5,620	3,548	1.58×
patricia	1,996	1,672	1.19×	1,804	984	1.83×
qsort-small	3,804	1,492	2.55×	2,432	840	2.90×
rsynth-tiny	109,864	15,320	7.17×	13,504	3,140	4.30×
search-large	840	740	1.14×	668	340	1.96×
search-small	828	728	1.14×	708	312	2.27×
sha	1,160	660	1.76×	1,032	448	2.30×
ssusan	6,608	1,824	3.62×	2,452	1,060	2.31×

Conclusion

- Novel cache design dedicated to stack data
- Analyzable caching strategy
- Does not require analysis of individual accesses
- Simple analysis
  - Compute displacement on call graph
  - Perform function-local data-flow analysis
  - Compute context-sensitive information 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