Efficient Context Switching for the Stack Cache: Implementation and Analysis

Sahar Abbaspour
Compute
DTU

Florian Brandner
LTCI, CNRS
Telecom ParisTech

Amine Naji
U2IS
ENSTA ParisTech

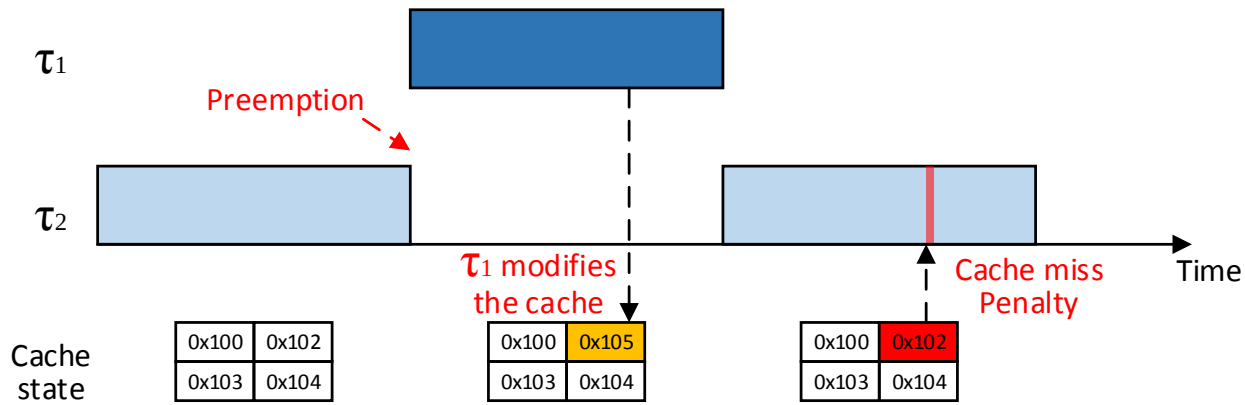
Mathieu Jan
CEA, LIST

This work is supported by the Digiteo project PM-TOP.

digiteo
Research in science and technology of information

Cache Related Preemption Delay - On Standard Caches

Cache Related Preemption Delay (CRPD): Time penalty introduced by cache misses due to task preemption.



- Preempting task may evict cache blocks of a preempted task.
- Cache misses may occur when the preempted task is resumed.

What is a Stack Cache?

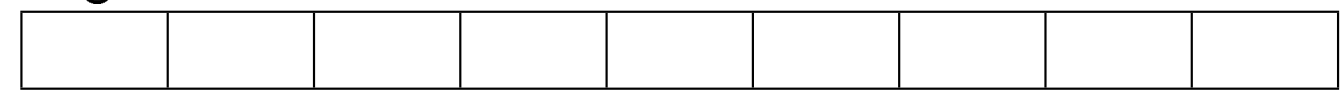
Dedicated cache for stack data

- Simple ring buffer
- 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

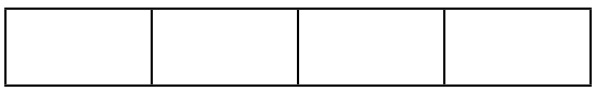
```
function A()        function B()        function C()
sres 2              sres 2              sres 3
call B()            call C()            sfree 3
sens 2              sens 2
sfree 2             sfree 2
```

Logical stack



MT ST

Stack cache*



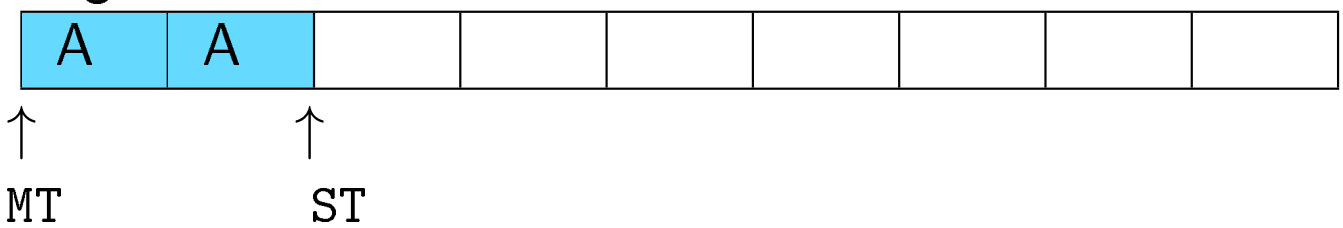
Occupancy = 0

*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



Stack cache*

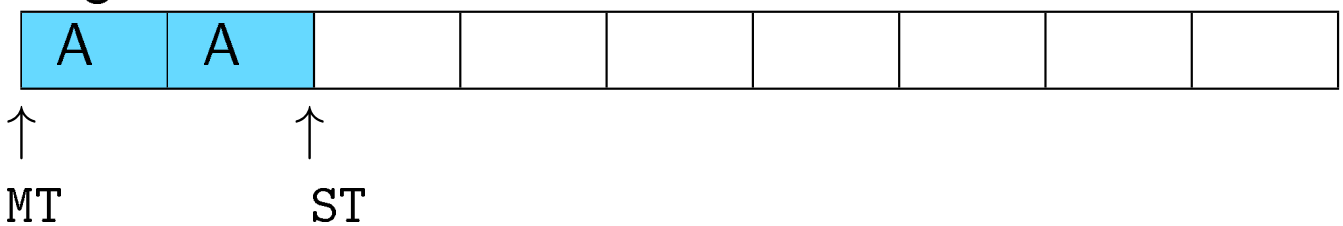
Occupancy = 2

*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



Stack cache*

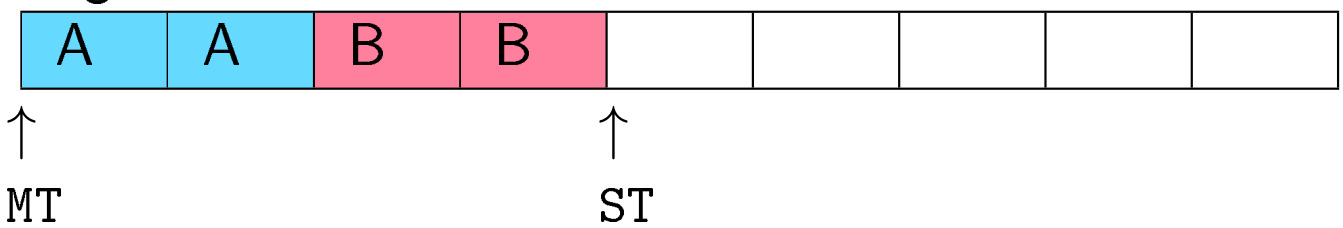
Occupancy = 2

*Cache configuration: 4 blocks

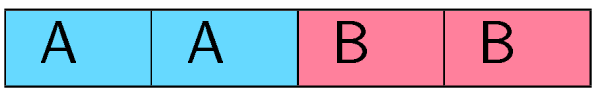
Example: Stack Cache

```
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



Stack cache*



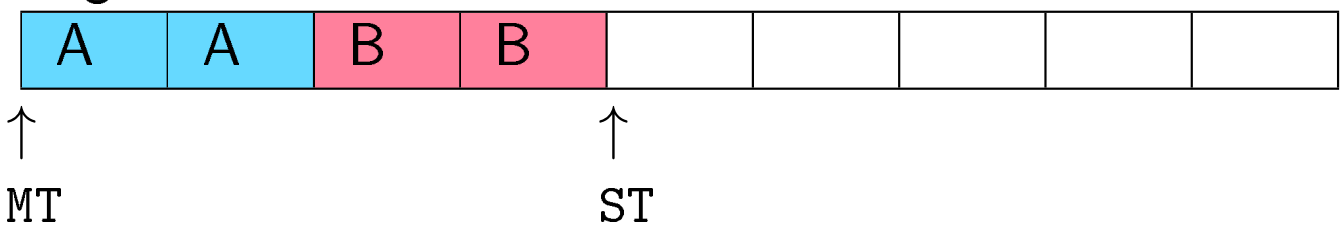
Occupancy = 4

*Cache configuration: 4 blocks

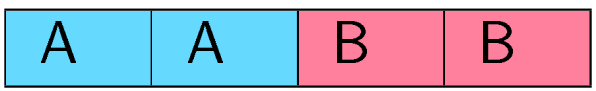
Example: Stack Cache

```
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



Stack cache*



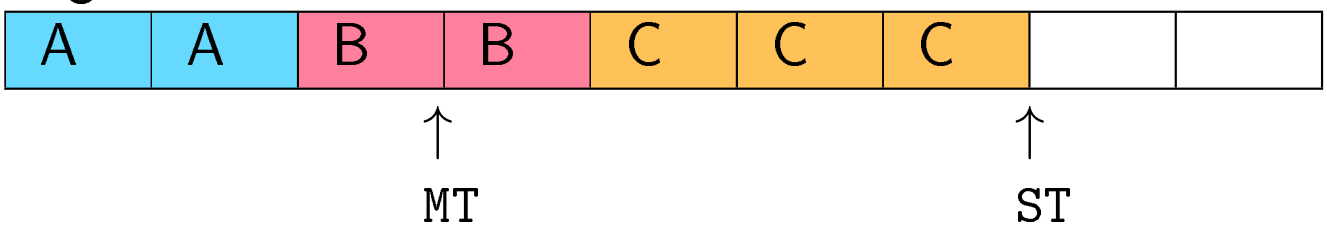
Occupancy = 4

*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



Stack cache*

Spill 3 blocks

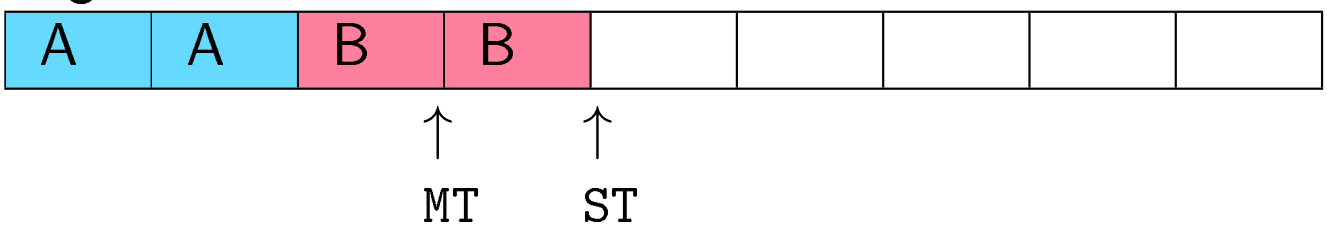
Occupancy = 4

*Cache configuration: 4 blocks

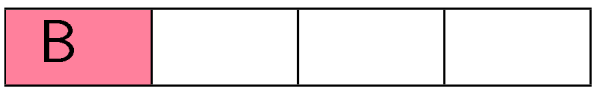
Example: Stack Cache

```
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



Stack cache*



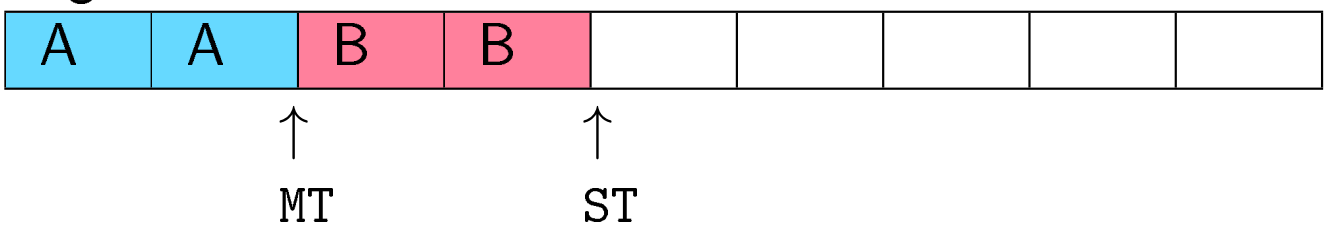
Occupancy = 1

*Cache configuration: 4 blocks

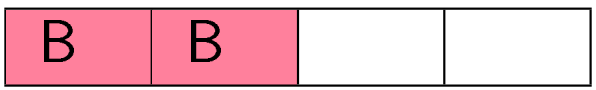
Example: Stack Cache

```
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



Stack cache*



Fill 1 block

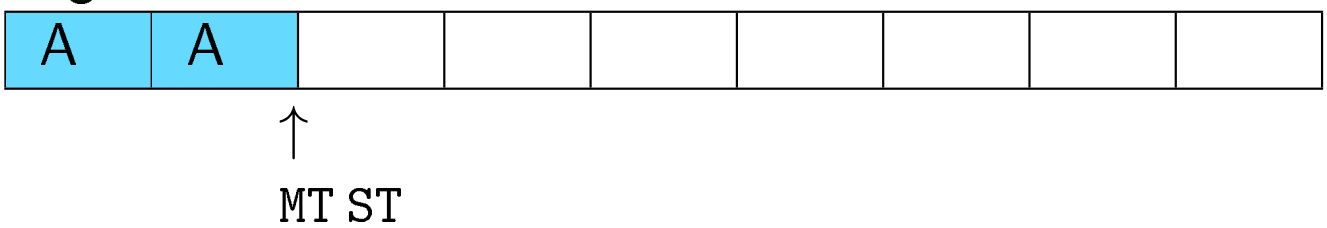
Occupancy = 2

*Cache configuration: 4 blocks

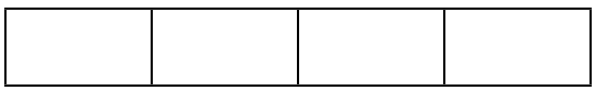
Example: Stack Cache

```
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



Stack cache*



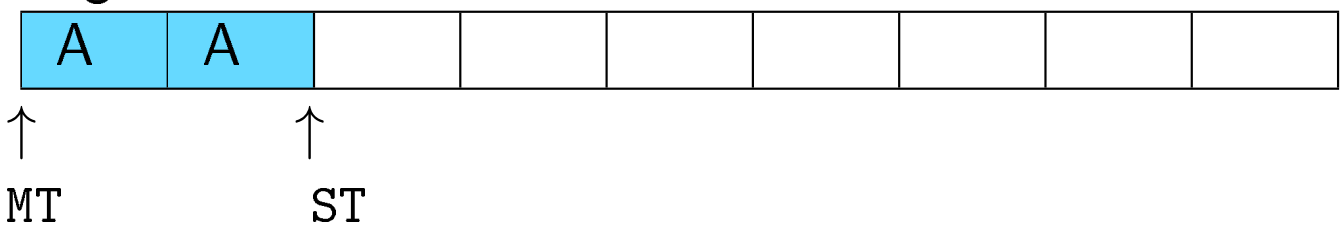
Occupancy = 0

*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



Stack cache*

Fill 2 blocks

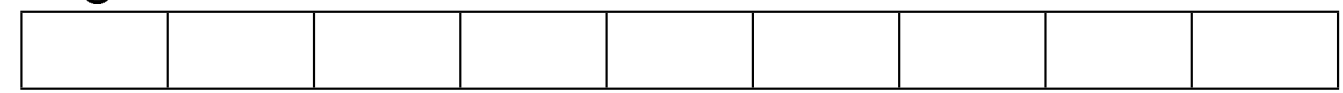
Occupancy = 2

*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
```

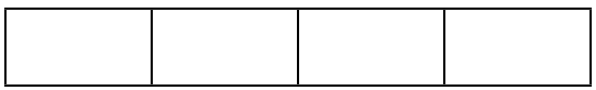
Logical stack



↑

MT ST

Stack cache*



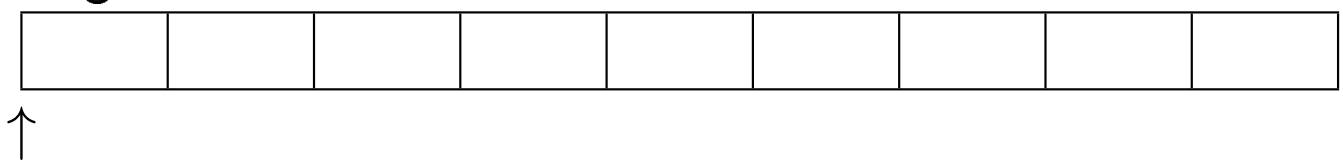
Occupancy = 0

*Cache configuration: 4 blocks

Example: Stack Cache

```
function A()        function B()        function C()
sres 2 ⟨0⟩          sres 2 ⟨0⟩          sres 3 ⟨3⟩
call B()            call C()            sfree 3
sens 2 ⟨2⟩          sens 2 ⟨1⟩
sfree 2             sfree 2
```

Logical stack



MT ST

Stack cache*

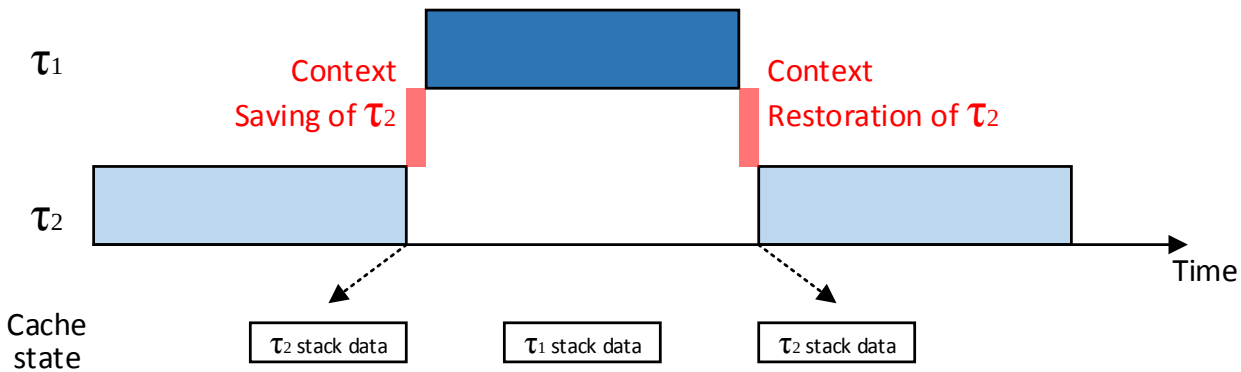
Bounds ⟨#⟩ provided by *standard* Stack Cache Analysis (no preemption)

*Cache configuration: 4 blocks

Context Switching - On Stack Cache

Due to its simplicity, the stack cache cannot be shared among tasks.

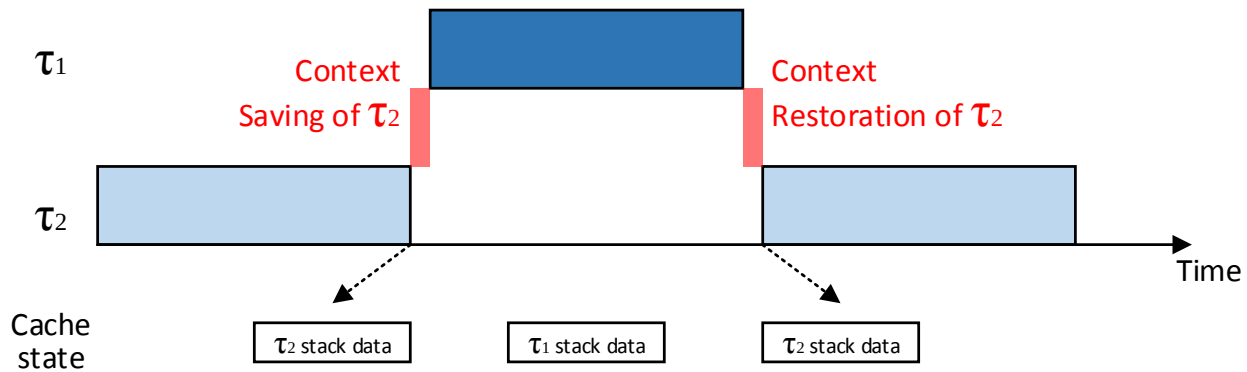
$\Rightarrow$ Context of preempted task has to be saved/restored.



Context Switching - On Stack Cache

Due to its simplicity, the stack cache cannot be shared among tasks.

$\Rightarrow$ Context of preempted task has to be saved/restored.



Main questions:

- Which cache blocks have to be saved/restored?
- What is the impact on the worst-case behavior after the preemption?

Motivational Example

```
(i_1^A) func A()
(i_2^A)  sres 2 ⟨0⟩
(i_3^A)  B()
(i_4^A)  sens 2 ⟨2⟩
(i_5^A)  sfree 2
```

```
(i_1^B) func B()
(i_2^B)  sres 2 ⟨0⟩
(i_3^B)  nop ↯
(i_4^B)  C()
(i_5^B)  sens 2 ⟨1⟩
(i_6^B)  sfree 2
```

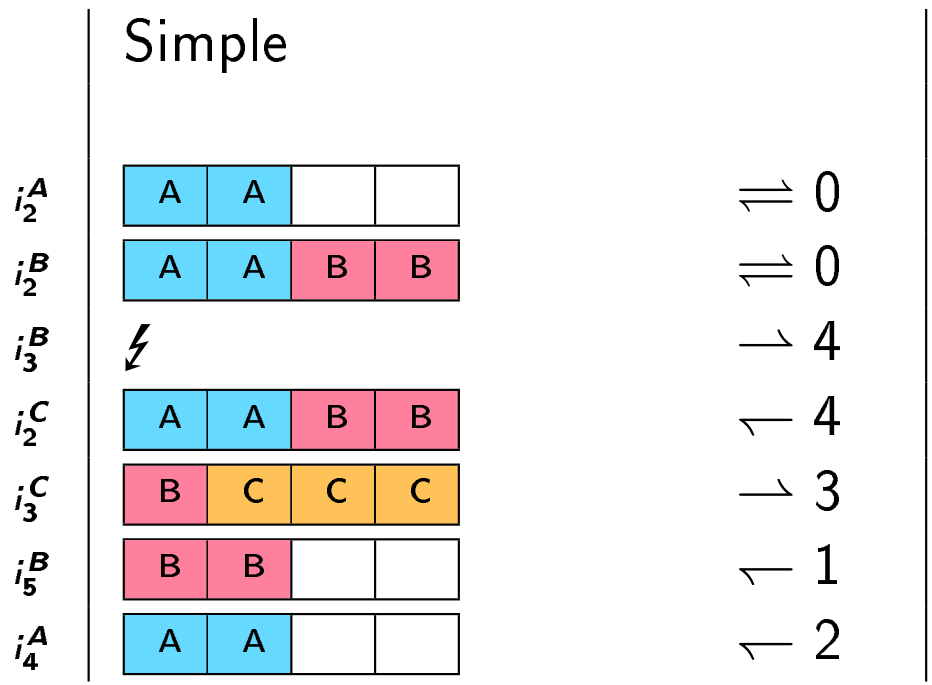
```
(i_1^C) func C()
(i_2^C)  sres 3 ⟨3⟩
(i_3^C)  sfree 3
```

Motivational Example

```
(i1A) func A()
(i2A)   sres 2 ⟨0⟩
(i3A)   B()
(i4A)   sens 2 ⟨2⟩
(i5A)   sfree 2
```

```
(i1B) func B()
(i2B)   sres 2 ⟨0⟩
(i3B)   nop ↯
(i4B)   C()
(i5B)   sens 2 ⟨1⟩
(i6B)   sfree 2
```

```
(i1C) func C()
(i2C)   sres 3 ⟨3⟩
(i3C)   sfree 3
```

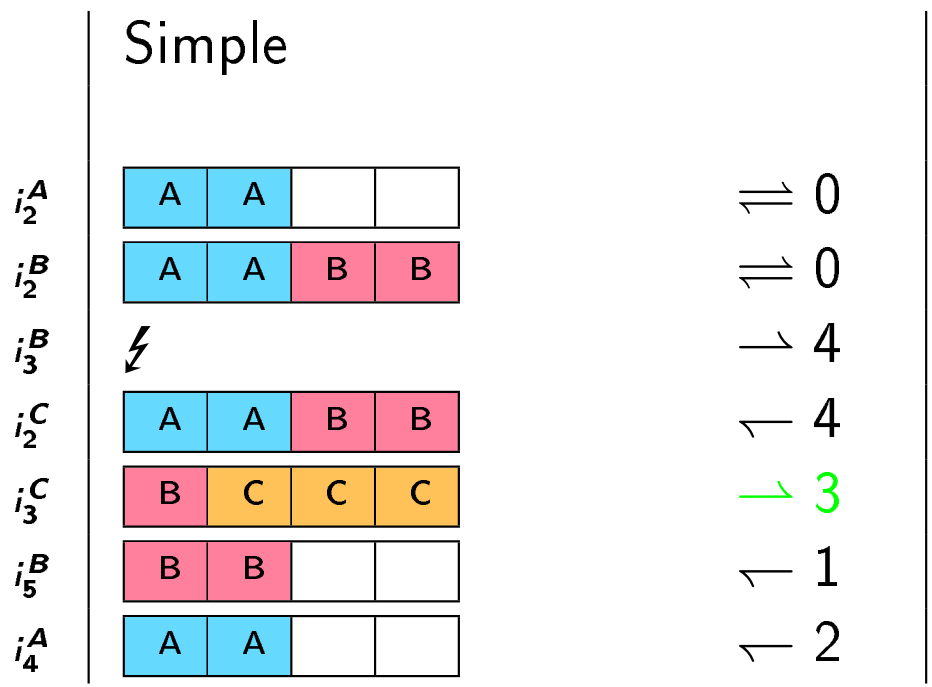


Motivational Example

```
(i1A) func A()
(i2A)   sres 2 ⟨0⟩
(i3A)   B()
(i4A)   sens 2 ⟨2⟩
(i5A)   sfree 2
```

```
(i1B) func B()
(i2B)   sres 2 ⟨0⟩
(i3B)   nop ↯
(i4B)   C()
(i5B)   sens 2 ⟨1⟩
(i6B)   sfree 2
```

```
(i1C) func C()
(i2C)   sres 3 ⟨3⟩
(i3C)   sfree 3
```

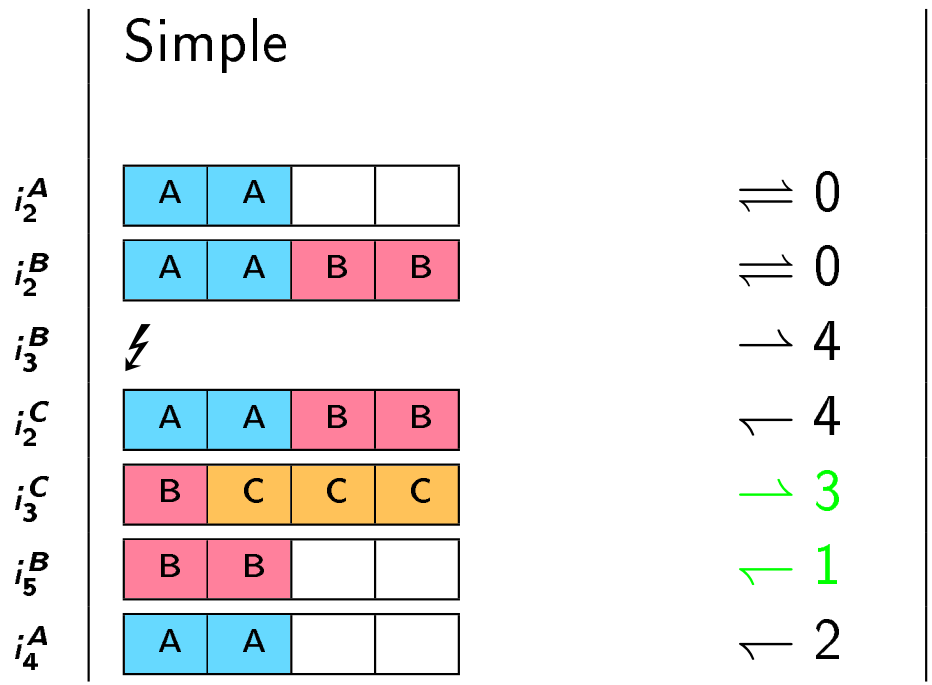


Motivational Example

```
(i1A) func A()
(i2A)   sres 2 ⟨0⟩
(i3A)   B()
(i4A)   sens 2 ⟨2⟩
(i5A)   sfree 2
```

```
(i1B) func B()
(i2B)   sres 2 ⟨0⟩
(i3B)   nop ↯
(i4B)   C()
(i5B)   sens 2 ⟨1⟩
(i6B)   sfree 2
```

```
(i1C) func C()
(i2C)   sres 3 ⟨3⟩
(i3C)   sfree 3
```

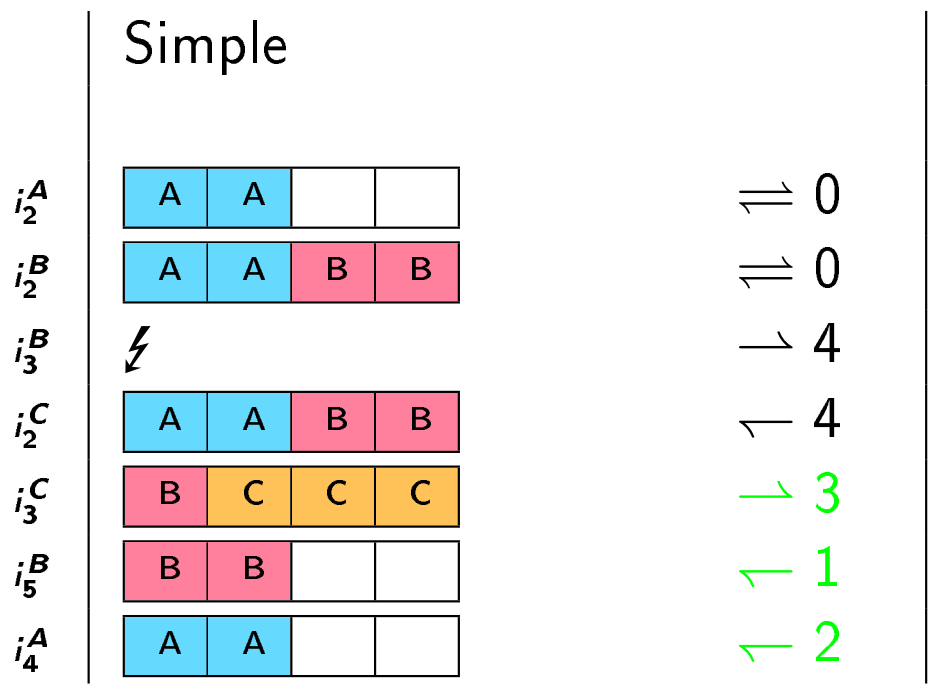


Motivational Example

```
(i1A) func A()
(i2A)   sres 2 ⟨0⟩
(i3A)   B()
(i4A)   sens 2 ⟨2⟩
(i5A)   sfree 2
```

```
(i1B) func B()
(i2B)   sres 2 ⟨0⟩
(i3B)   nop ↯
(i4B)   C()
(i5B)   sens 2 ⟨1⟩
(i6B)   sfree 2
```

```
(i1C) func C()
(i2C)   sres 3 ⟨3⟩
(i3C)   sfree 3
```

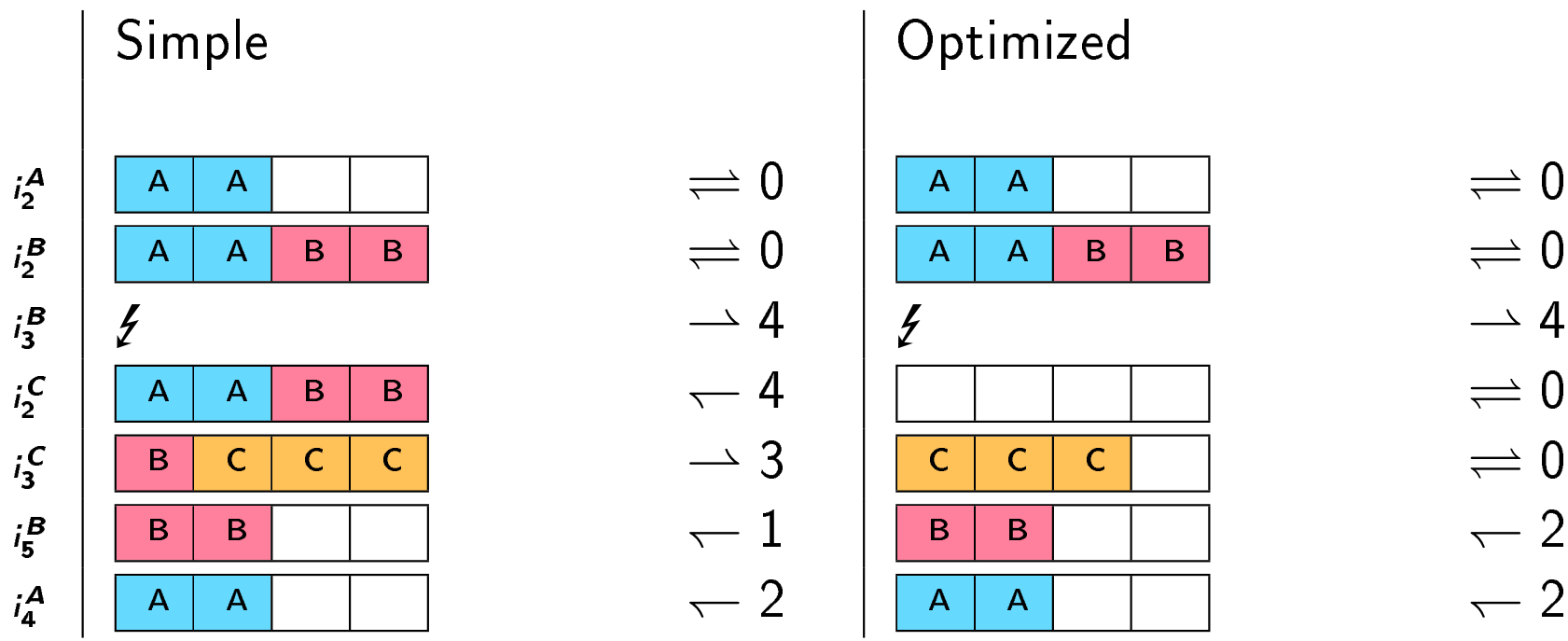


Motivational Example

```
(i1A) func A()
(i2A)   sres 2 ⟨0⟩
(i3A)   B()
(i4A)   sens 2 ⟨2⟩
(i5A)   sfree 2
```

```
(i1B) func B()
(i2B)   sres 2 ⟨0⟩
(i3B)   nop ↯
(i4B)   C()
(i5B)   sens 2 ⟨1⟩
(i6B)   sfree 2
```

```
(i1C) func C()
(i2C)   sres 3 ⟨3⟩
(i3C)   sfree 3
```

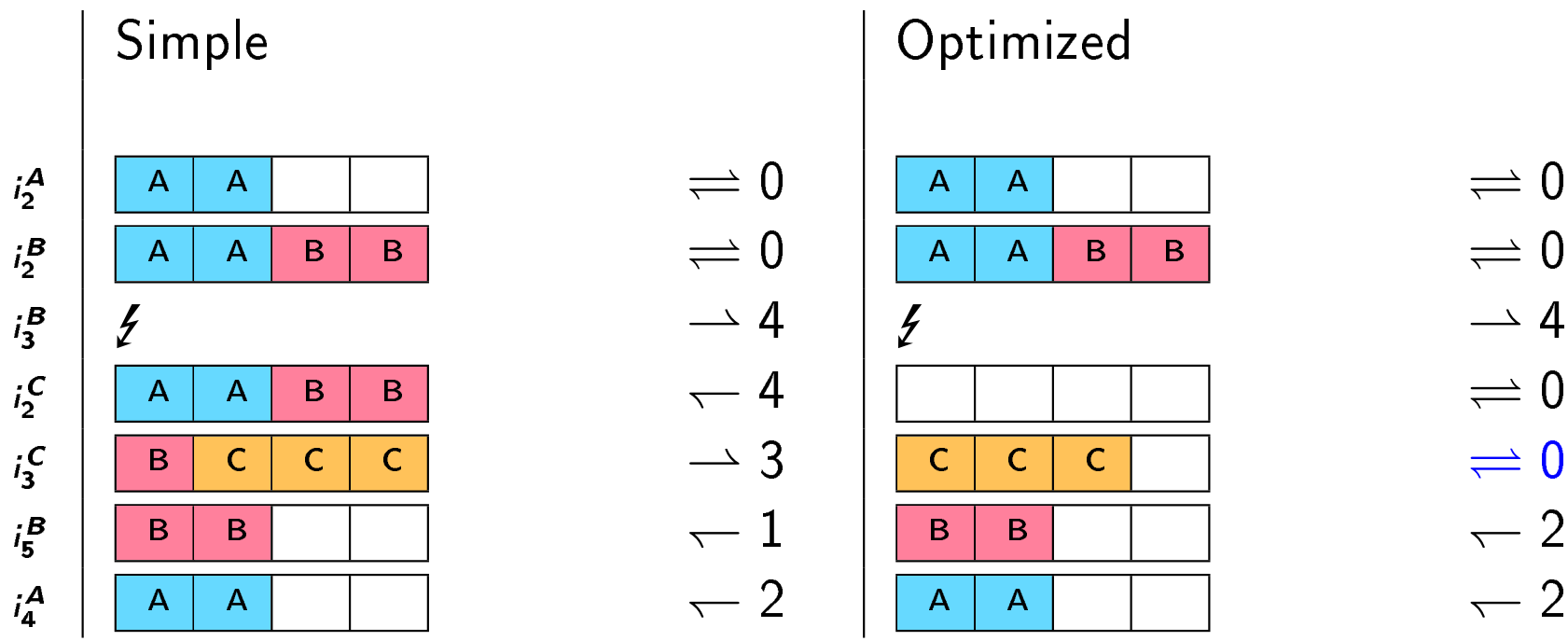


Motivational Example

```
(i1A) func A()
(i2A)   sres 2 ⟨0⟩
(i3A)   B()
(i4A)   sens 2 ⟨2⟩
(i5A)   sfree 2
```

```
(i1B) func B()
(i2B)   sres 2 ⟨0⟩
(i3B)   nop ↯
(i4B)   C()
(i5B)   sens 2 ⟨1⟩
(i6B)   sfree 2
```

```
(i1C) func C()
(i2C)   sres 3 ⟨3⟩
(i3C)   sfree 3
```

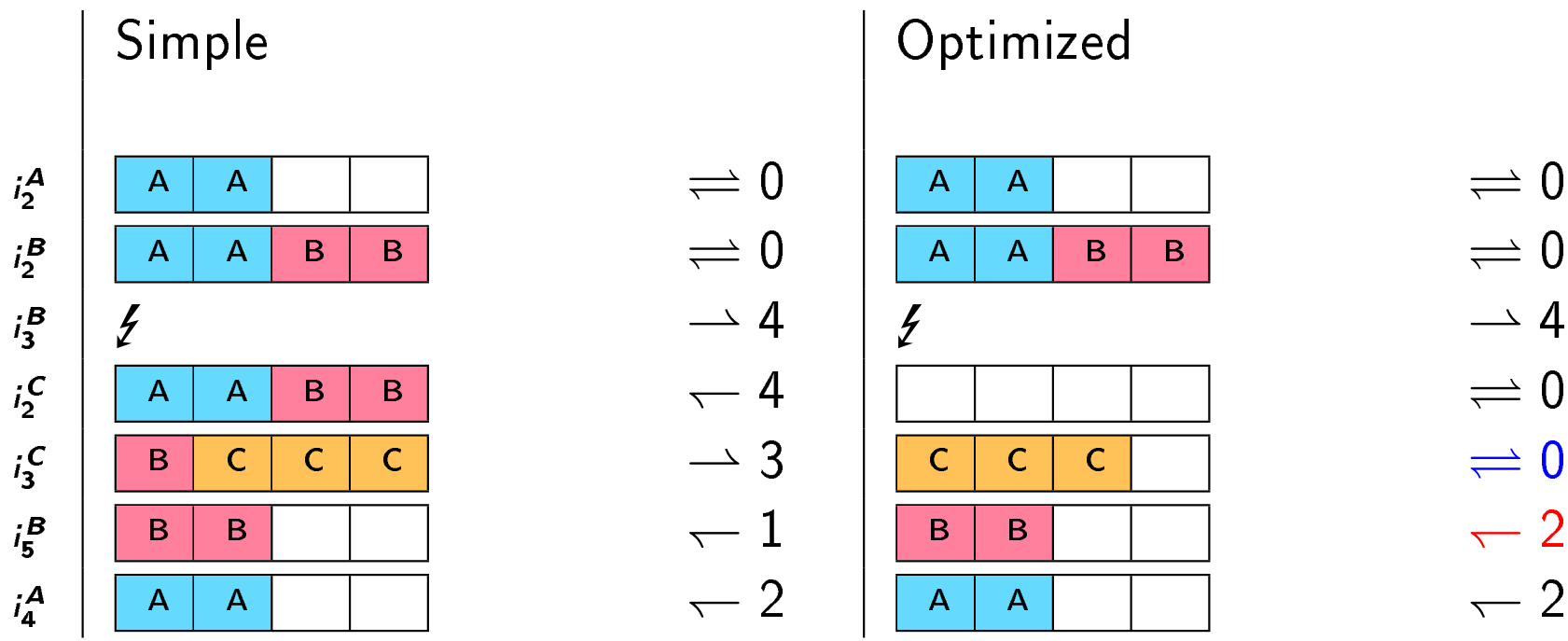


Motivational Example

```
(i1A) func A()
(i2A)   sres 2 ⟨0⟩
(i3A)   B()
(i4A)   sens 2 ⟨2⟩
(i5A)   sfree 2
```

```
(i1B) func B()
(i2B)   sres 2 ⟨0⟩
(i3B)   nop ↯
(i4B)   C()
(i5B)   sens 2 ⟨1⟩
(i6B)   sfree 2
```

```
(i1C) func C()
(i2C)   sres 3 ⟨3⟩
(i3C)   sfree 3
```

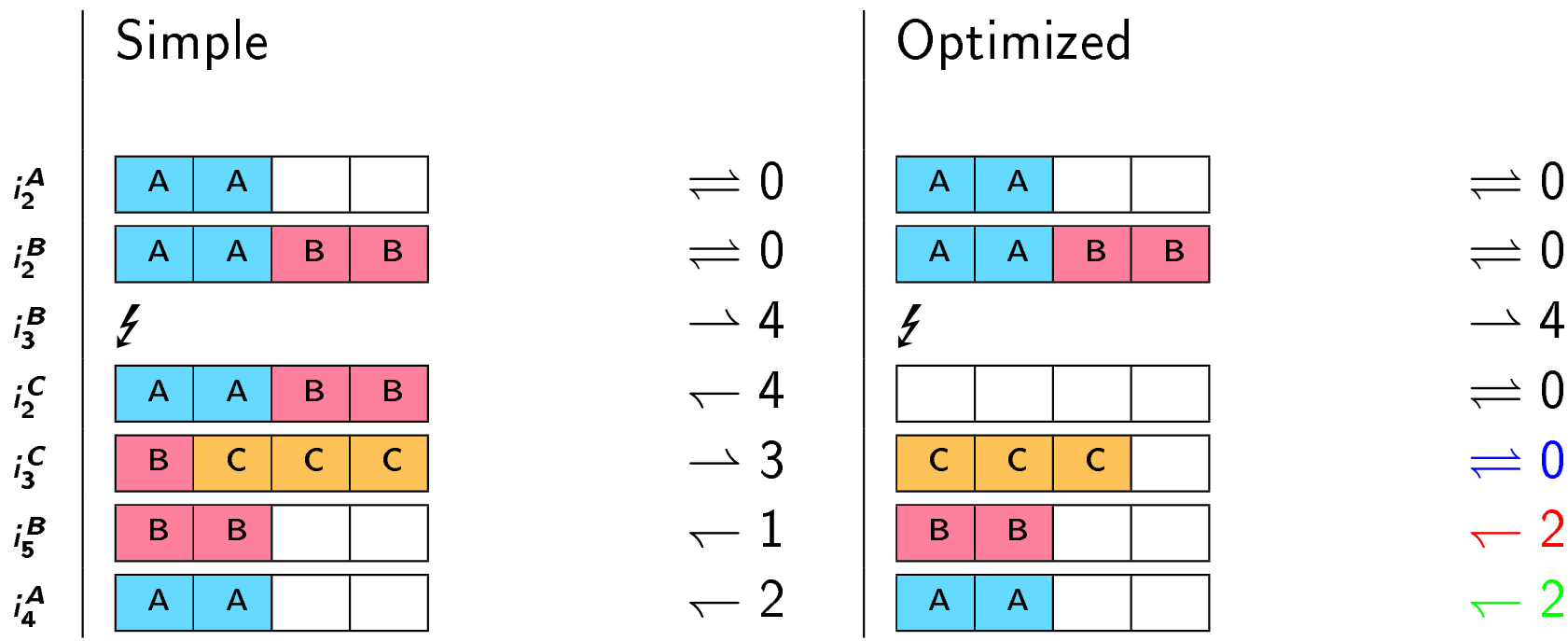


Motivational Example

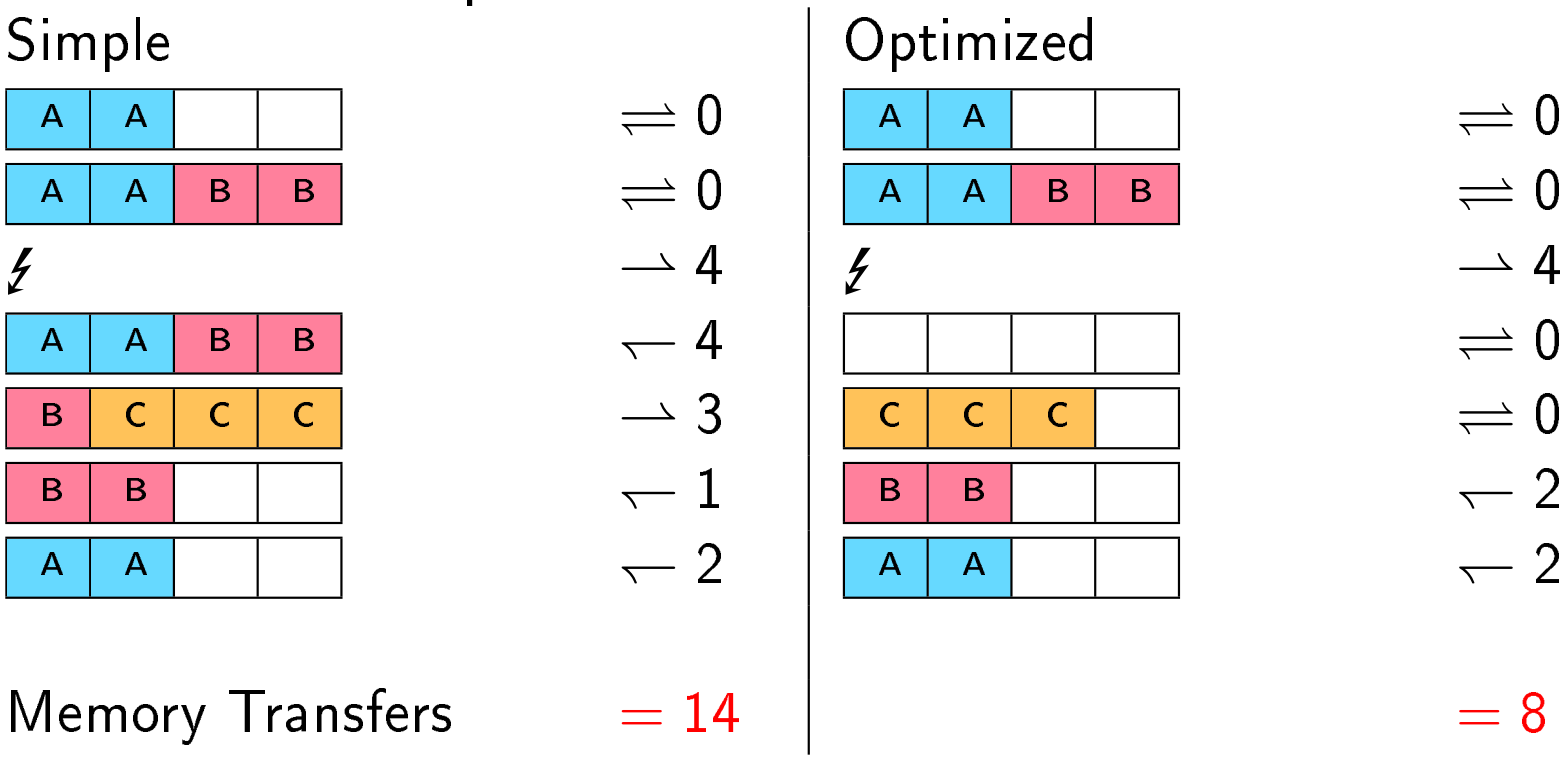
```
(i1A) func A()
(i2A)   sres 2 ⟨0⟩
(i3A)   B()
(i4A)   sens 2 ⟨2⟩
(i5A)   sfree 2
```

```
(i1B) func B()
(i2B)   sres 2 ⟨0⟩
(i3B)   nop ↯
(i4B)   C()
(i5B)   sens 2 ⟨1⟩
(i6B)   sfree 2
```

```
(i1C) func C()
(i2C)   sres 3 ⟨3⟩
(i3C)   sfree 3
```



Motivational Example



Optimized approach : Potentially fewer memory transfers, but worst-case behavior has to be re-analyzed.

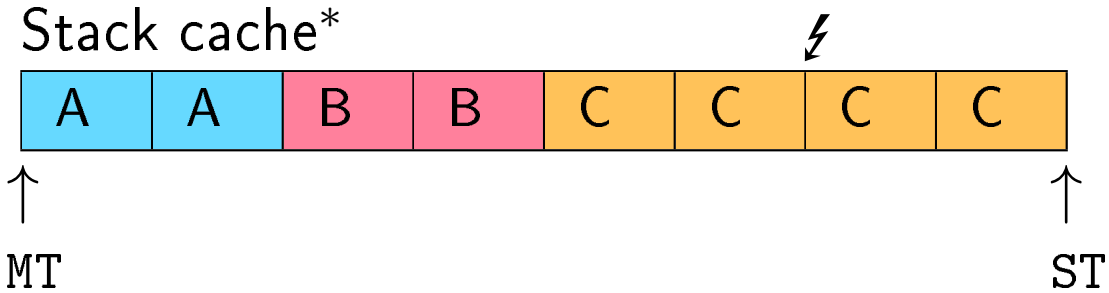
Two analysis problems :

- Context Saving Analysis (CSA)
- Context Restoring Analysis (CRA)

Context Saving Analysis (CSA)

Split the stack cache into three regions by introducing two pointers:

- Lazy Pointer (LP) keeps track of coherent data.
  - Analysis introduced in previous work.
- Dead Pointer (DP) keeps track of certainly dead data.
  - Using data-flow analysis (livness analysis).

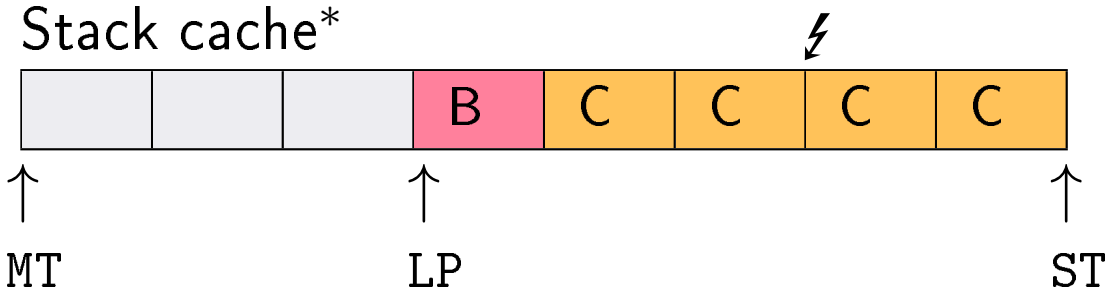


*Cache configuration: 8 blocks

Context Saving Analysis (CSA)

Split the stack cache into three regions by introducing two pointers:

- Lazy Pointer (LP) keeps track of coherent data.
  - Analysis introduced in previous work.
- Dead Pointer (DP) keeps track of certainly dead data.
  - Using data-flow analysis (livness analysis).

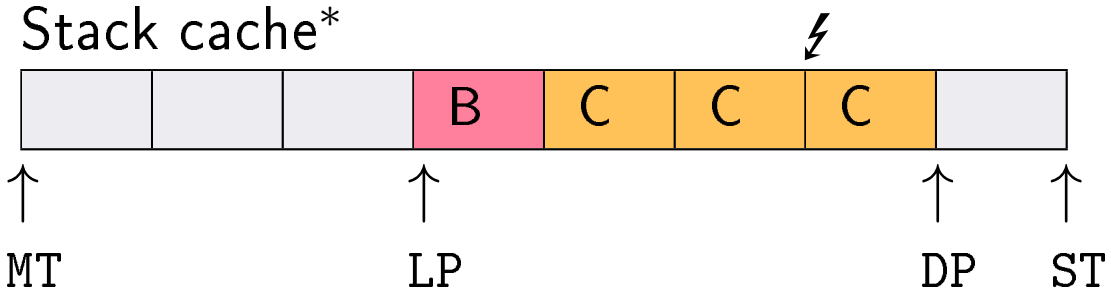


*Cache configuration: 8 blocks

Context Saving Analysis (CSA)

Split the stack cache into three regions by introducing two pointers:

- Lazy Pointer (LP) keeps track of coherent data.
  - Analysis introduced in previous work.
- Dead Pointer (DP) keeps track of certainly dead data.
  - Using data-flow analysis (livness analysis).

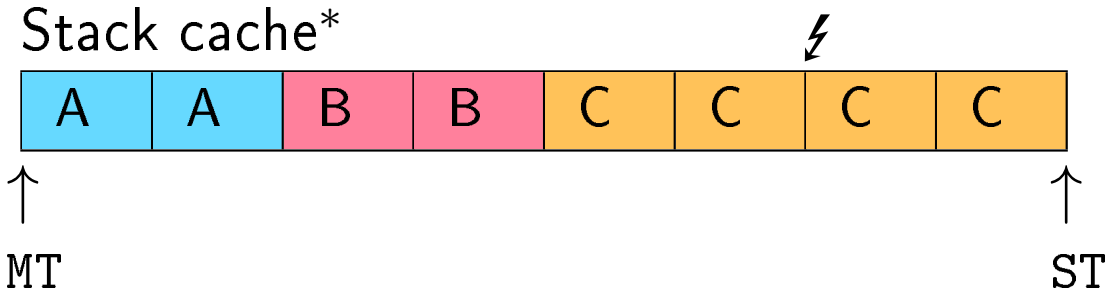


*Cache configuration: 8 blocks

Context Restoring Analysis (CRA)

Split the stack cache into three regions introducing two pointers.

- Restore Pointer (RP) keeps track of data that has to be restored explicitly. Excluding :
  - Stack frames of calling functions.
  - Local data that will be ensured.
  - Analyzed using data-flow analysis (similar to liveness)
- Global analysis of gains at reserves.
- Global analysis of additional costs at ensures.

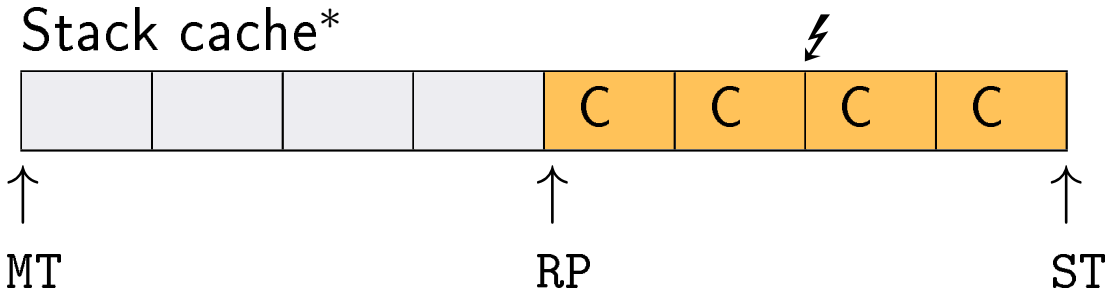


*Cache configuration: 8 blocks

Context Restoring Analysis (CRA)

Split the stack cache into three regions introducing two pointers.

- Restore Pointer (RP) keeps track of data that has to be restored explicitly. Excluding :
  - Stack frames of calling functions.
  - Local data that will be ensured.
  - Analyzed using data-flow analysis (similar to liveness)
- Global analysis of gains at reserves.
- Global analysis of additional costs at ensures.

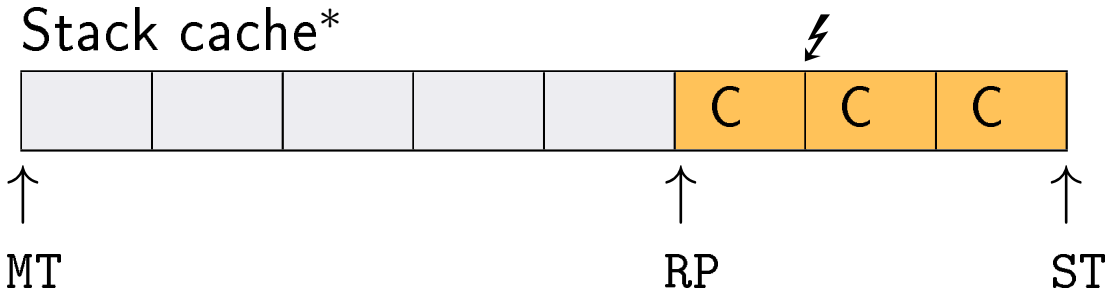


*Cache configuration: 8 blocks

Context Restoring Analysis (CRA)

Split the stack cache into three regions introducing two pointers.

- Restore Pointer (RP) keeps track of data that has to be restored explicitly. Excluding :
  - Stack frames of calling functions.
  - Local data that will be ensured.
  - Analyzed using data-flow analysis (similar to liveness)
- Global analysis of gains at reserves.
- Global analysis of additional costs at ensures.

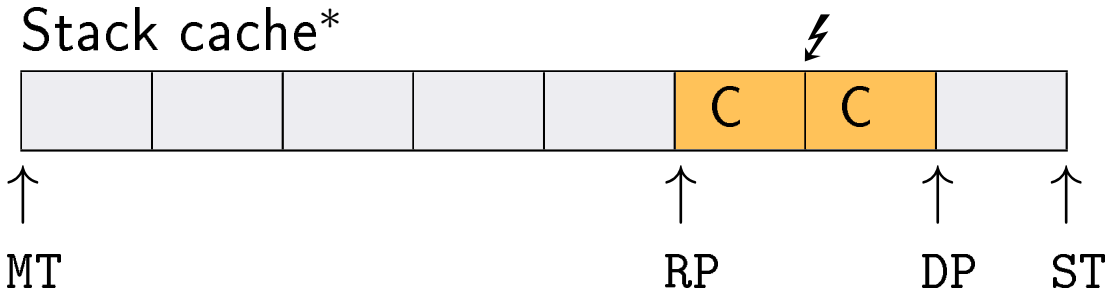


*Cache configuration: 8 blocks

Context Restoring Analysis (CRA)

Split the stack cache into three regions introducing two pointers.

- Restore Pointer (RP) keeps track of data that has to be restored explicitly. Excluding :
  - Stack frames of calling functions.
  - Local data that will be ensured.
  - Analyzed using data-flow analysis (similar to liveness)
- Global analysis of gains at reserves.
- Global analysis of additional costs at ensures.

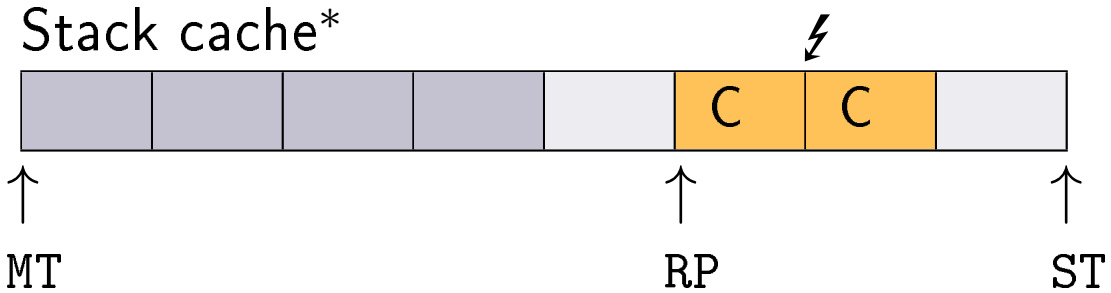


*Cache configuration: 8 blocks

Context Restoring Analysis (CRA)

Split the stack cache into three regions introducing two pointers.

- Restore Pointer (RP) keeps track of data that has to be restored explicitly. Excluding :
  - Stack frames of calling functions.
  - Local data that will be ensured.
  - Analyzed using data-flow analysis (similar to liveness)
- Global analysis of gains at reserves.
- Global analysis of additional costs at ensures.



*Cache configuration: 8 blocks

Global Cost of Ensure Instruction

- `sens` instructions partially restore stack frames for free*.
- We need to account for the additional costs.

```
(i1^A) func A()
(i2^A)   sres 2 ⟨0⟩
(i3^A)   B()
(i4^A)   sens 2 ⟨2⟩
(i5^A)   sfree 2
```

```
(i1^B) func B()
(i2^B)   sres 2 ⟨0⟩
(i4^B)   C()
(i5^B)   sens 2 ⟨1⟩
(i6^B)   sfree 2
```

```
(i1^C) func C()
(i2^C)   sres 3 ⟨3⟩
(i3^C)   nop ↯
(i4^C)   sfree 3
```

*For free : already taken into account by the standard Stack Cache Analysis.

Global Analysis of Ensure Cost - Example

```
(i_1^A) func A()
(i_2^A)   sres 2 ⟨0⟩
(i_3^A)   B()
(i_4^A)   sens 2 ⟨2⟩
(i_5^A)   sfree 2
```

```
(i_1^B) func B()
(i_2^B)   sres 2 ⟨0⟩
(i_4^B)   C()
(i_5^B)   sens 2 ⟨1⟩
(i_6^B)   sfree 2
```

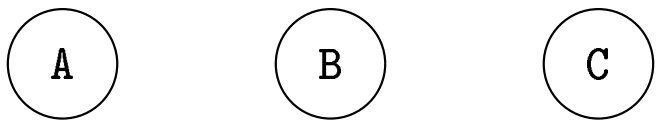
```
(i_1^C) func C()
(i_2^C)   sres 3 ⟨3⟩
(i_3^C)   nop ↯
(i_4^C)   sfree 3
```

Global Analysis of Ensure Cost - Example

```
(i_1^A) func A()
(i_2^A)  sres 2 ⟨0⟩
(i_3^A)  B()
(i_4^A)  sens 2 ⟨2⟩
(i_5^A)  sfree 2
```

```
(i_1^B) func B()
(i_2^B)  sres 2 ⟨0⟩
(i_4^B)  C()
(i_5^B)  sens 2 ⟨1⟩
(i_6^B)  sfree 2
```

```
(i_1^C) func C()
(i_2^C)  sres 3 ⟨3⟩
(i_3^C)  nop ↯
(i_4^C)  sfree 3
```

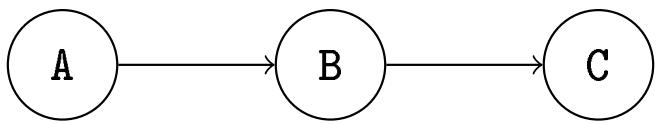


Global Analysis of Ensure Cost - Example

```
(i1A) func A()
(i2A)  sres 2 ⟨0⟩
(i3A)  B()
(i4A)  sens 2 ⟨2⟩
(i5A)  sfree 2
```

```
(i1B) func B()
(i2B)  sres 2 ⟨0⟩
(i4B)  C()
(i5B)  sens 2 ⟨1⟩
(i6B)  sfree 2
```

```
(i1C) func C()
(i2C)  sres 3 ⟨3⟩
(i3C)  nop ↯
(i4C)  sfree 3
```

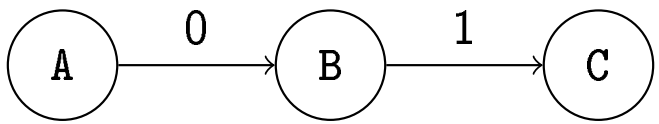


Global Analysis of Ensure Cost - Example

```
(i1^A) func A()
(i2^A)   sres 2 ⟨0⟩
(i3^A)   B()
(i4^A)   sens 2 ⟨2⟩
(i5^A)   sfree 2
```

```
(i1^B) func B()
(i2^B)   sres 2 ⟨0⟩
(i4^B)   C()
(i5^B)   sens 2 ⟨1⟩
(i6^B)   sfree 2
```

```
(i1^C) func C()
(i2^C)   sres 3 ⟨3⟩
(i3^C)   nop ↯
(i4^C)   sfree 3
```

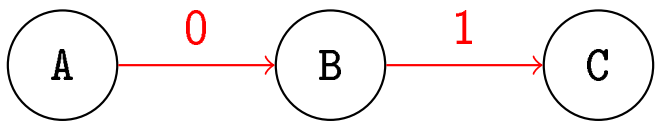


Global Analysis of Ensure Cost - Example

```
(i1A) func A()
(i2A)   sres 2 ⟨0⟩
(i3A)   B()
(i4A)   sens 2 ⟨2⟩
(i5A)   sfree 2
```

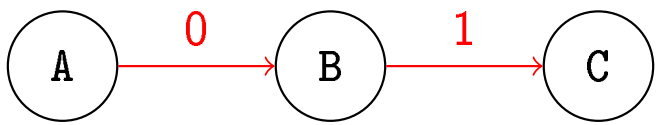
```
(i1B) func B()
(i2B)   sres 2 ⟨0⟩
(i4B)   C()
(i5B)   sens 2 ⟨1⟩
(i6B)   sfree 2
```

```
(i1C) func C()
(i2C)   sres 3 ⟨3⟩
(i3C)   nop ↯
(i4C)   sfree 3
```



Global Analysis of Ensure Cost - Example

```
(i1^A) func A()          (i1^B) func B()          (i1^C) func C()
(i2^A)   sres 2 ⟨0⟩      (i2^B)   sres 2 ⟨0⟩      (i2^C)   sres 3 ⟨3⟩
(i3^A)   B()             (i4^B)   C()            (i3^C)   nop ↯
(i4^A)   sens 2 ⟨2⟩      (i5^B)   sens 2 ⟨1⟩      (i4^C)   sfree 3
(i5^A)   sfree 2         (i6^B)   sfree 2
```



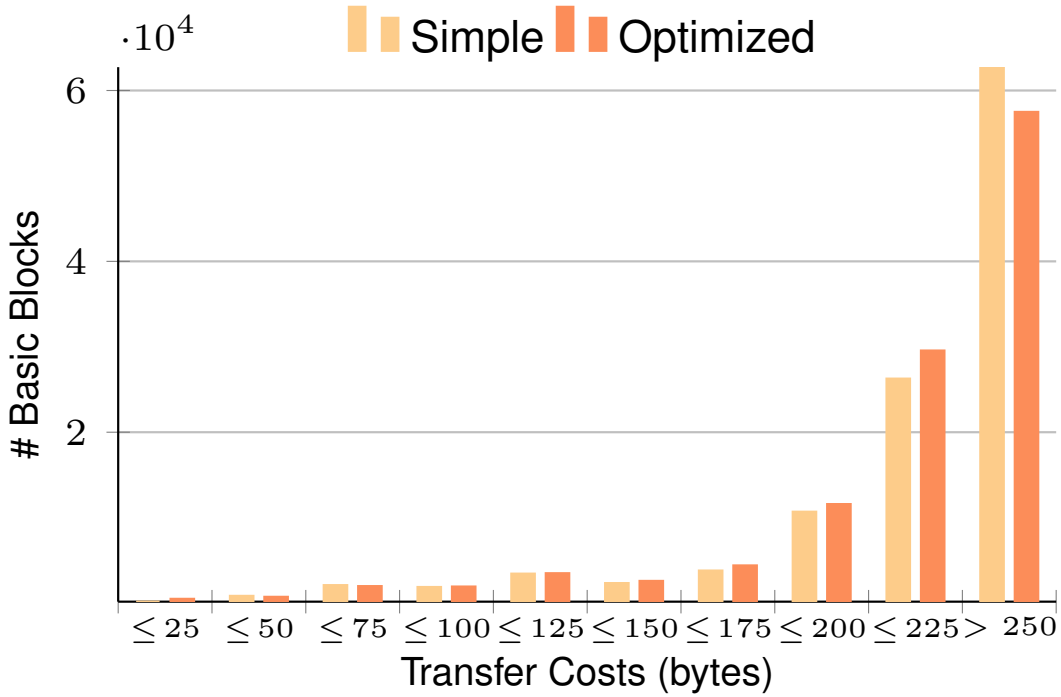
Observation

The length of the path is bounded by the stack cache size.

Experimental Setup

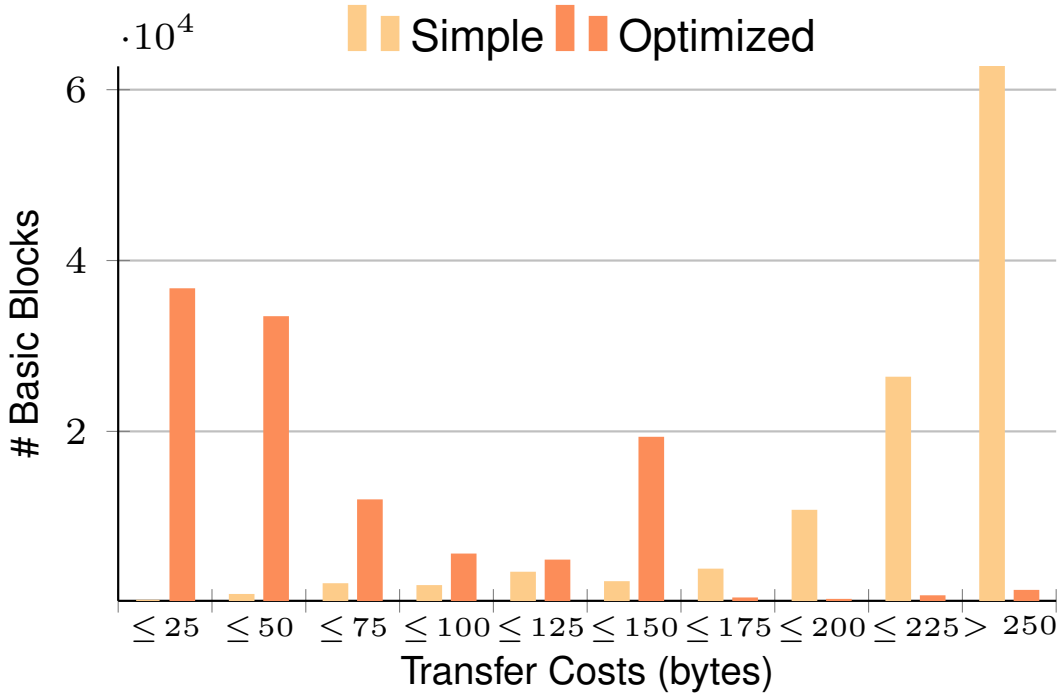
- MiBench benchmark suite
- LLVM compiler 3.5 for the Patmos processor
- Compiled with optimizations enabled (-O2)
- Stack cache configurations: 256B
- Compile benchmarks and analyze preemption costs

Experiments: Context Saving Analysis



- Improvement in around 16% of basic blocks
- Shift from right to the left

Experiments: Context Restoring Analysis



- Drastic shift from right to the left.
- Improvement in about 99% of basic blocks.
- In many cases (43.4%) no explicit memory transfer is needed.

Also in the paper...

Detailed context saving/restoring cost analyses.

- Data-flow equations controlling DP, LP, RP pointers.
- Analysis of local ensures costs.
- Analysis reserve gain.

Virtual Stack Cache: first step to integrate our analysis into schedulability analysis.

Conclusion

- We proposed a preemption cost analysis for the stack cache.
  - Context Saving Analysis.
  - Context Restoring Analysis.
- Low complexity due to function-local data-flow analyses.
- Inter-procedural effects are handled through variants of the longest path problem.
- Our analysis outperform the simple approach in most cases.

For future work:

- New task model exploiting information given by the analysis.
- Prefetching techniques to perform context saving/restoring using Virtual Stack Cache.

Thanks for your attention

Any Questions ?