Preemption Delay Analysis for the Stack Cache

Amine Naji
U2IS
ENSTA ParisTech

Florian Brandner
LTCI, CNRS
Telecom ParisTech

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

Real-Time Systems

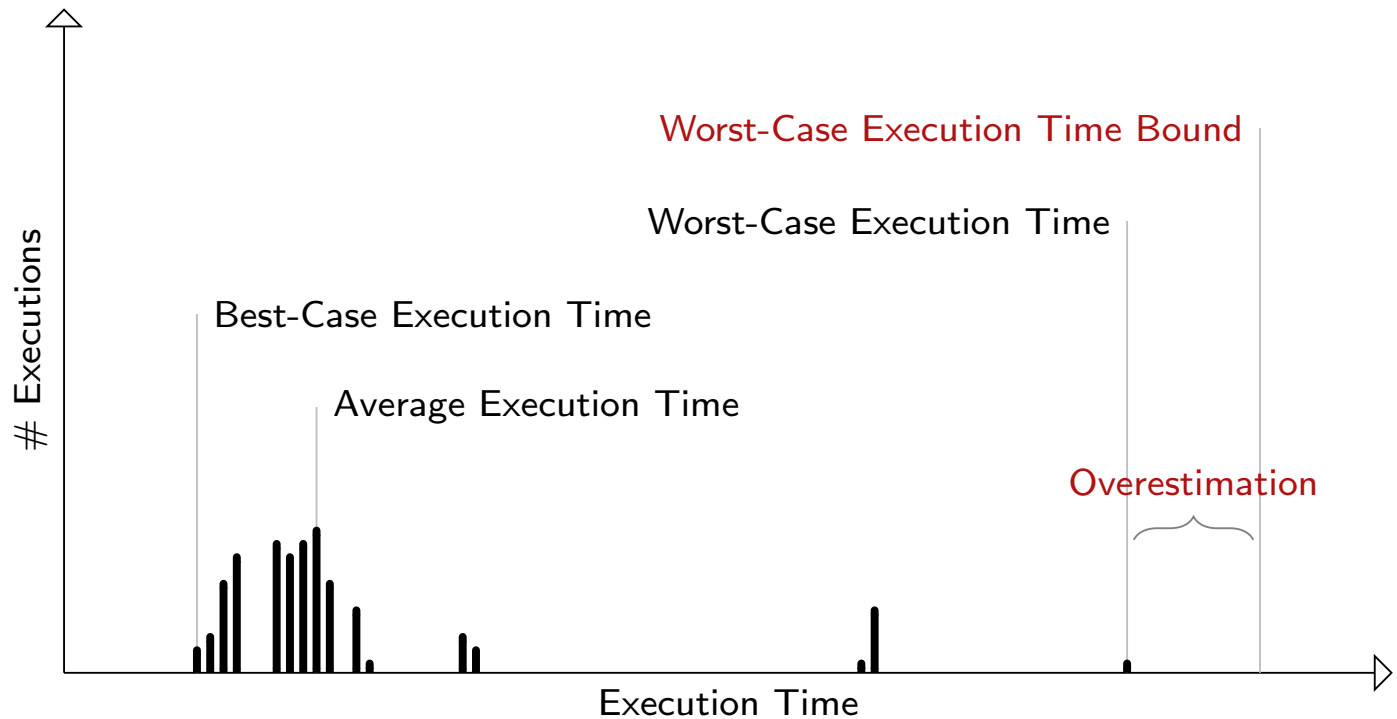
Strict timing **guarantees**

- Critical tasks have to be completed in time

Real-Time Systems

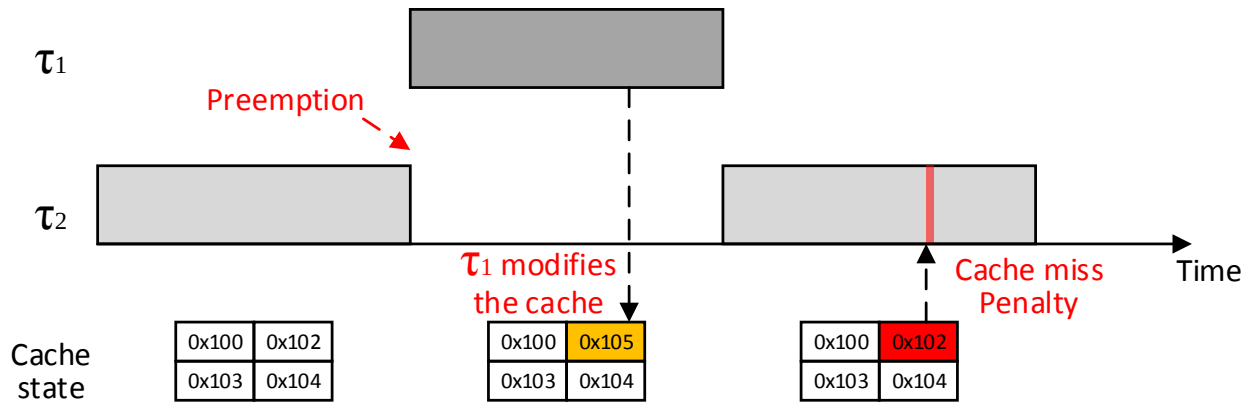
Strict timing **guarantees**

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



Cache Related Preemption Delay - On Standard Caches

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



- Some cache blocks of a 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?

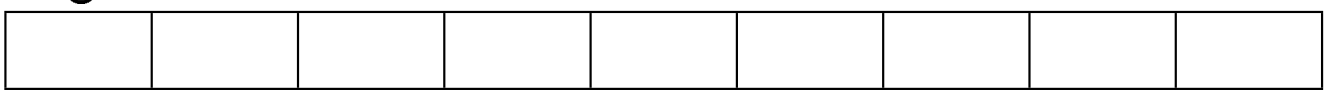
Dedicated cache for stack data

- Simple ring buffer (*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

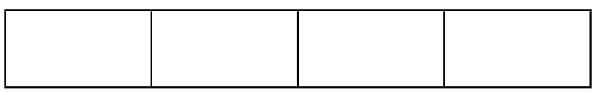
Example: Stack Cache

```
(1) function A()        function B()        function C()
(2) sres 2 <0>          sres 2 <0>          sres 3 <3>
(3) call B()            call C()            sfree 3
(4) sens 2 <2>          sens 2 <1>
(5) sfree 2             sfree 2
```

Logical stack



Stack cache*

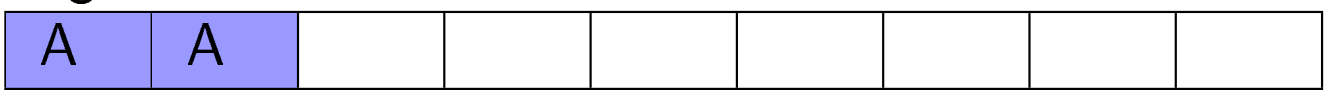


*Cache configuration: 4 blocks

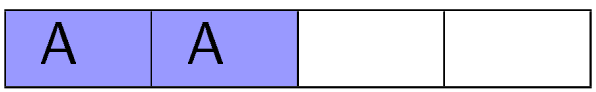
Example: Stack Cache

```
(1) function A()        function B()        function C()
(2) sres 2 <0> ←        sres 2 <0>          sres 3 <3>
(3) call B()            call C()            sfree 3
(4) sens 2 <2>          sens 2 <1>
(5) sfree 2             sfree 2
```

Logical stack



Stack cache*

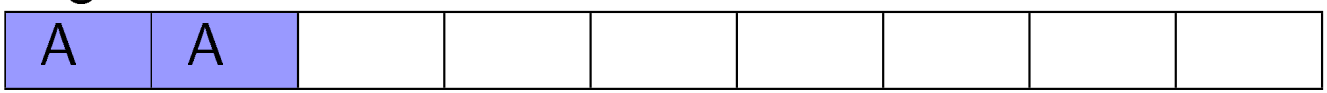


*Cache configuration: 4 blocks

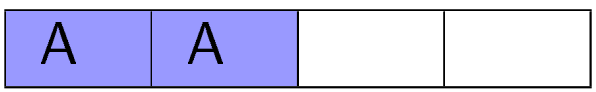
Example: Stack Cache

```
(1) function A()        function B()        function C()
(2) sres 2 <0>          sres 2 <0>          sres 3 <3>
(3) call B() ←          call C()            sfree 3
(4) sens 2 <2>          sens 2 <1>
(5) sfree 2             sfree 2
```

Logical stack



Stack cache*

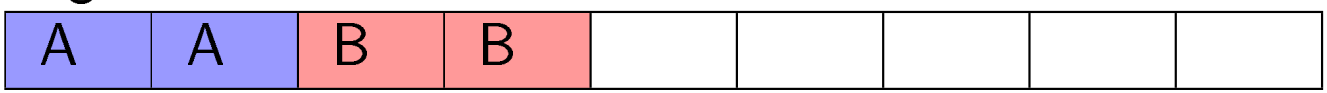


*Cache configuration: 4 blocks

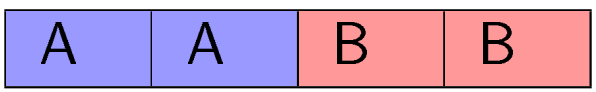
Example: Stack Cache

```
(1) function A()        function B()          function C()
(2) sres 2 <0>          sres 2 <0> ←          sres 3 <3>
(3) call B()            call C()              sfree 3
(4) sens 2 <2>          sens 2 <1>
(5) sfree 2             sfree 2
```

Logical stack



Stack cache*

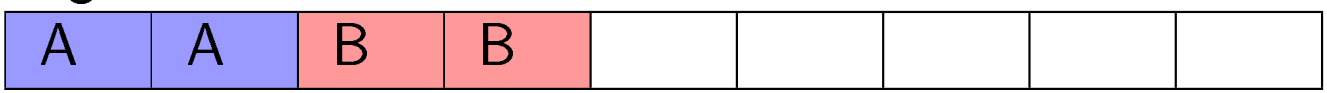


*Cache configuration: 4 blocks

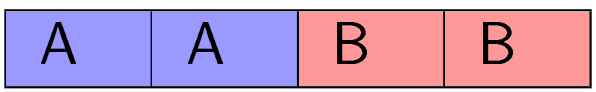
Example: Stack Cache

```
(1) function A()        function B()        function C()
(2) sres 2 <0>          sres 2 <0>          sres 3 <3>
(3) call B()            call C() ←          sfree 3
(4) sens 2 <2>          sens 2 <1>
(5) sfree 2             sfree 2
```

Logical stack



Stack cache*

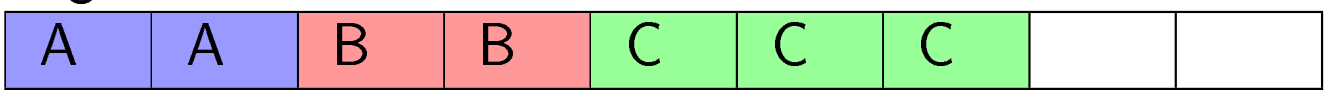


*Cache configuration: 4 blocks

Example: Stack Cache

```
(1) function A()      function B()      function C()
(2) sres 2 <0>        sres 2 <0>        sres 3 <3> ←
(3) call B()          call C()          sfree 3
(4) sens 2 <2>        sens 2 <1>
(5) sfree 2           sfree 2
```

Logical stack



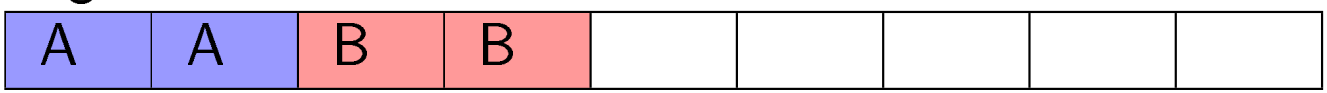
Stack cache*

*Cache configuration: 4 blocks

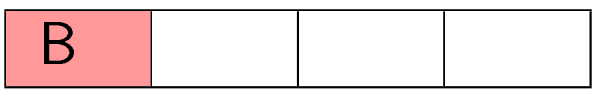
Example: Stack Cache

```
(1) function A()        function B()        function C()
(2) sres 2 <0>          sres 2 <0>          sres 3 <3>
(3) call B()            call C()            sfree 3 ←
(4) sens 2 <2>          sens 2 <1>
(5) sfree 2             sfree 2
```

Logical stack



Stack cache*

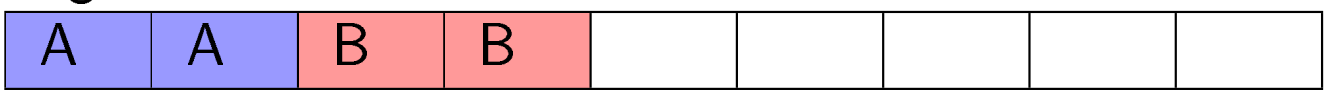


*Cache configuration: 4 blocks

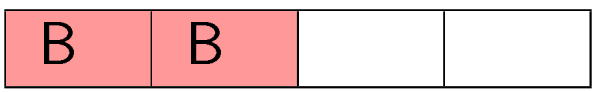
Example: Stack Cache

```
(1) function A()        function B()        function C()
(2) sres 2 <0>          sres 2 <0>          sres 3 <3>
(3) call B()            call C()            sfree 3
(4) sens 2 <2>          sens 2 <1> ←
(5) sfree 2             sfree 2
```

Logical stack



Stack cache*

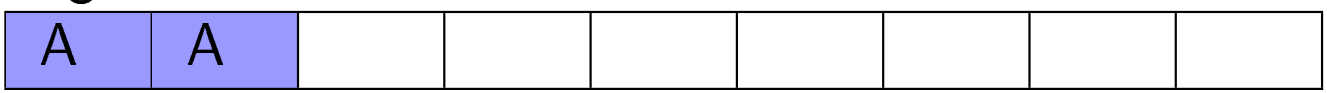


*Cache configuration: 4 blocks

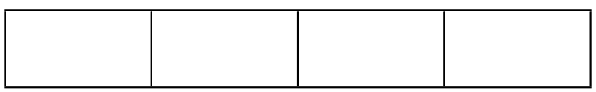
Example: Stack Cache

```
(1) function A()        function B()        function C()
(2) sres 2 <0>          sres 2 <0>          sres 3 <3>
(3) call B()            call C()            sfree 3
(4) sens 2 <2>          sens 2 <1>
(5) sfree 2             sfree 2 ←
```

Logical stack



Stack cache*

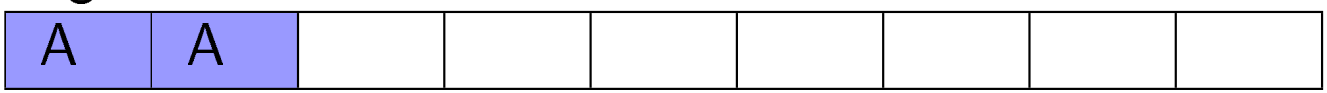


*Cache configuration: 4 blocks

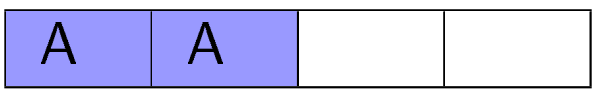
Example: Stack Cache

```
(1) function A()        function B()        function C()
(2) sres 2 <0>          sres 2 <0>          sres 3 <3>
(3) call B()            call C()            sfree 3
(4) sens 2 <2> ←        sens 2 <1>
(5) sfree 2             sfree 2
```

Logical stack



Stack cache*

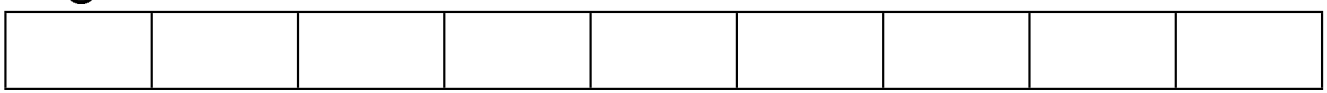


*Cache configuration: 4 blocks

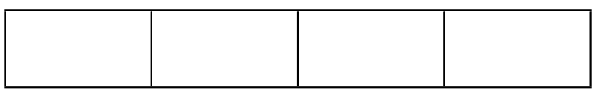
Example: Stack Cache

```
(1) function A()      function B()      function C()
(2) sres 2 <0>        sres 2 <0>        sres 3 <3>
(3) call B()          call C()          sfree 3
(4) sens 2 <2>        sens 2 <1>
(5) sfree 2 ←         sfree 2
```

Logical stack



Stack cache*



*Cache configuration: 4 blocks

Cache Related Preemption Delay - On Stack Cache

The original design of the stack cache did not consider the multitasking aspects.

The stack space cannot be shared with tasks.

Context of preempted task has to be saved/restored.

Two analysis problems:

- Context Saving
- Context Restoring

We seek to compute CRPD relative to the stack cache.

Preemption Cost Examples

```
(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
```

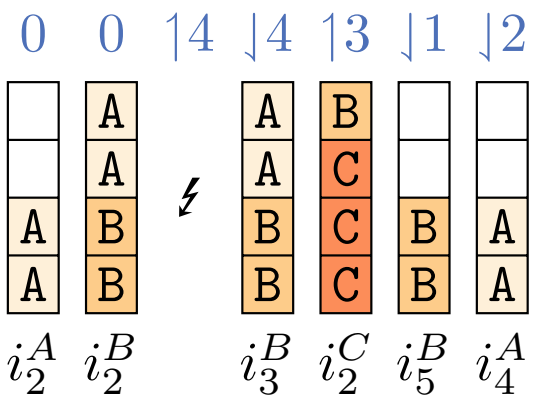
```
(i_1^C) func C()
(i_2^C)   sres 3 ⟨3⟩
(i_3^C)   sfree 3
```

Preemption Cost Examples

```
(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
```



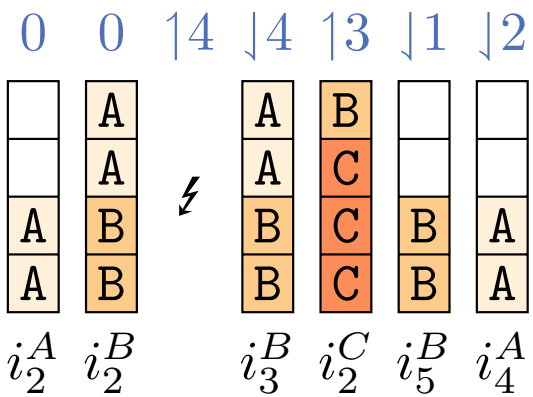
Simple approach.

Preemption Cost Examples

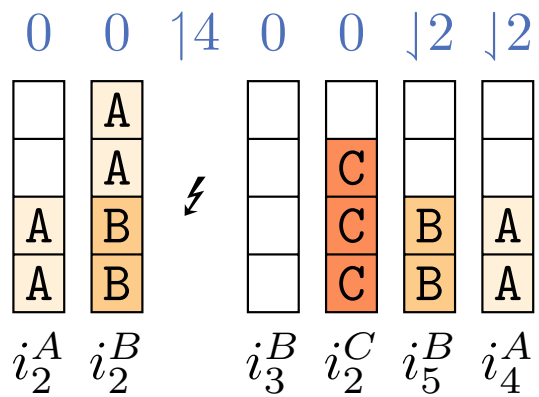
```
(i_1^A) func A()
(i_2^A)   sres 2 ⟨0⟩
(i_3^A)   B()
(i_4^A)   sens 2 ⟨2⟩
(i_5^A)   sfree 2
```

```
(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
```



Simple approach.



Improved approach.

Drawbacks of the Naïve Approach

Naive approach: Simple analysis, only based on the cache occupancy (originally provided by SCA analysis)

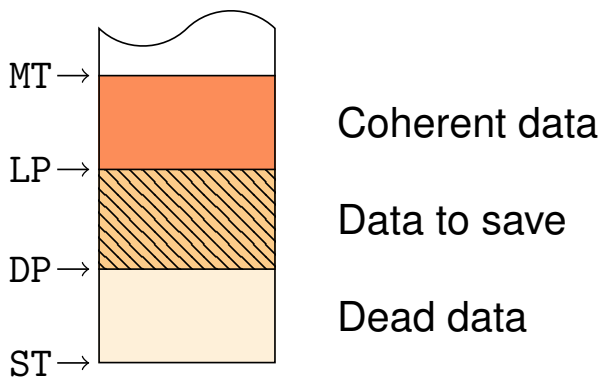
However, some inaccuracies may be introduced...

- Not all saved/restored data may be accessed afterwards.
- Unnecessary saving of coherent data to main memory.
- Analysis does not take advantage of placement of ensures

Context Saving Analysis - Overview

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

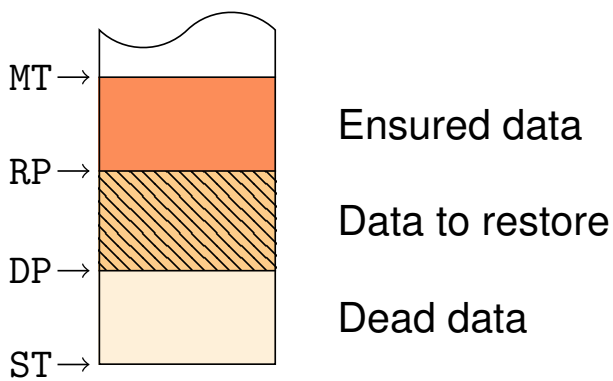
- Dead Pointer (DP) keeps track of dead data.
- Lazy Pointer (LP) keeps track of coherent data.
- Two pointer defines two areas, Dead Area is located below DP, while Coherent Area is located above LP.
- Regions size is computed using local data-flow analyses.



Context Restoring Analysis - Overview

Split the stack cache into three different regions introducing two pointers

- Dead data is not restored, and Restore Pointer (RP) keeps track of data that has to be restored explicitly.
- `sens` instruction will load the rest.
- Size of these regions is computed using function local data-flow analyses.
- Only the stack frame of the current function is restored
- Inter-procedural analysis accounts for additional overheads



Global Cost of Ensure Instruction

- Intuitively, `sens` instructions will partially restore stack frames for free. We need to account for the additional cost paid by `sens` instructions.

```
(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

- Costs can be derived from the longest path in a weighted CG from the program's entry node to the current function.
- Weights represents the difference between the corresponding ensure bounds and the calling function reserved size.
- Path length represent additional cost.

Global Analysis of Ensure Cost - Example

Global Ensure and Reserve Analyses - Example:

(i_1^A) `func A()`	(i_1^B) `func B()`	(i_1^C) `func C()`
(i_2^A) `sres 2` $\langle 0 \rangle$	(i_2^B) `sres 2` $\langle 0 \rangle$	(i_2^C) `sres 3` $\langle 3 \rangle$
(i_3^A) `B()`	(i_4^B) `C()`	(i_3^C) `nop` ↯
(i_4^A) `sens 2` $\langle 2 \rangle$	(i_5^B) `sens 2` $\langle 1 \rangle$	(i_4^C) `sfree 3`
(i_5^A) `sfree 2`	(i_6^B) `sfree 2`	

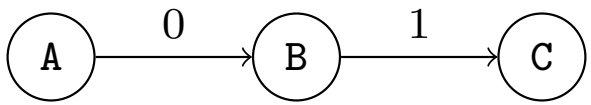
Global Analysis of Ensure Cost - Example

Global Ensure and Reserve Analyses - 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
```



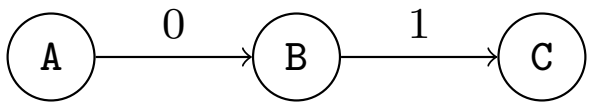
Global Analysis of Ensure Cost - Example

Global Ensure and Reserve Analyses - 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
```



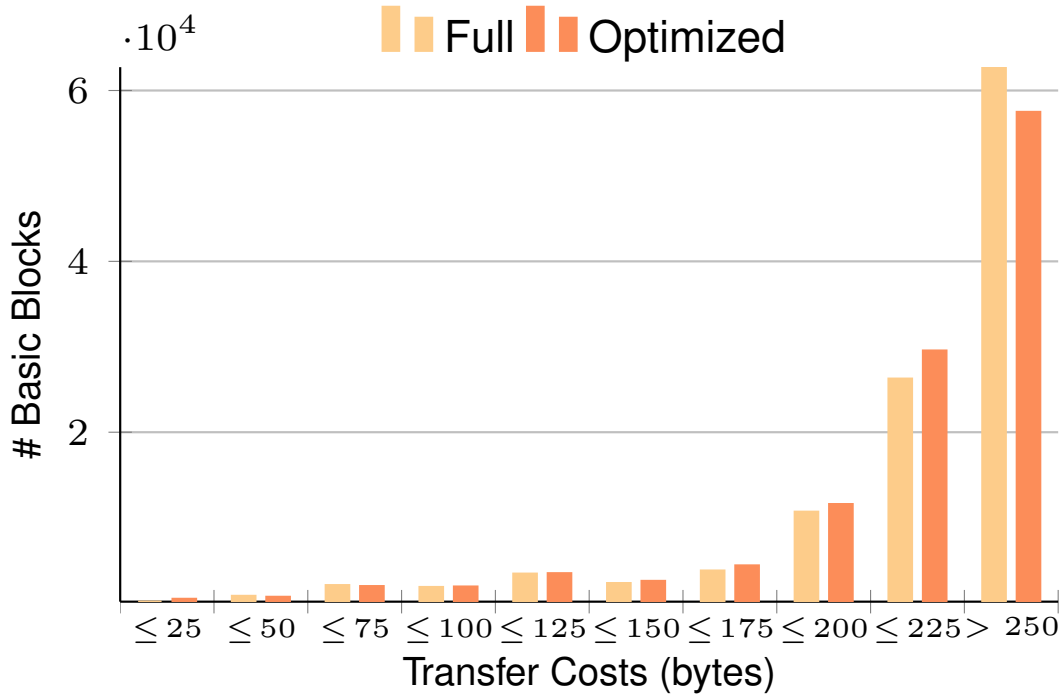
Observation

The length of the path is bounded by the stack cache size and the minimum amount of stack data remaining in the stack cache after returning from the function.

Experimental Setup

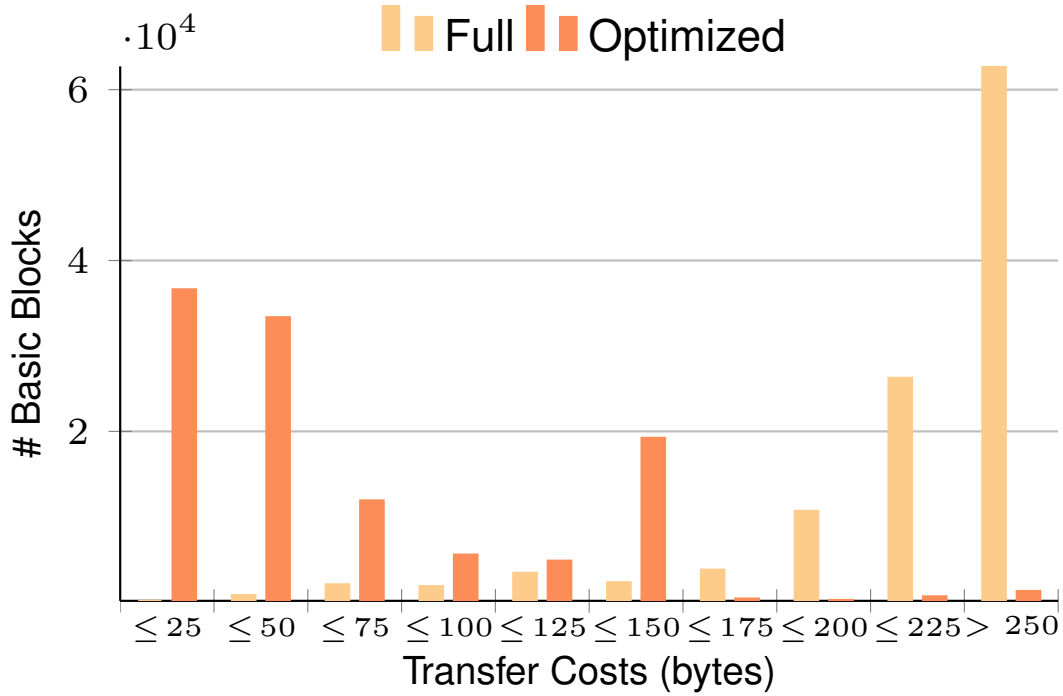
- MiBench benchmark suite
- LLVM compiler 3.5 for the Patmos processor
- Compiled with optimizations enabled (-O2)
- Stack cache configurations: 256B
- Compile benchmarks and perform preemption delay analyses

Experiments: Context Saving Analysis



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

Experiments: Context Restoring Analysis



- Drastic shift from right to the left
- Improvement in around 99% of basic blocks
- In some cases the program runs even faster (1.7%)

Conclusion

- We proposed a static analysis to determine preemption delay associated with the stack cache.
- Several function-local data-flow analysis
- Inter-procedural effects are handled through variants of the longest path problem.

Thanks for your attention

Any Question ?