Abstract interpretation of memory models

Pietro Ferrara

ETH Zurich
Switzerland

IFIP WG 2.3 Programming Methodology, Lachen, Switzerland

Multithreading

"(…) in order for an application to take advantage of the dual-core capabilities, the application should be optimized for multithreading."

G. Koch. Discovering multi-core: extending the benefits of Moore's law. In Technology Intel Magazine. Intel, July 2005.

- Parallelism supported through multithreading
  > Java and C#
- Implicit communications via shared memory
- Synchronization on monitors
- Subtle and problematic

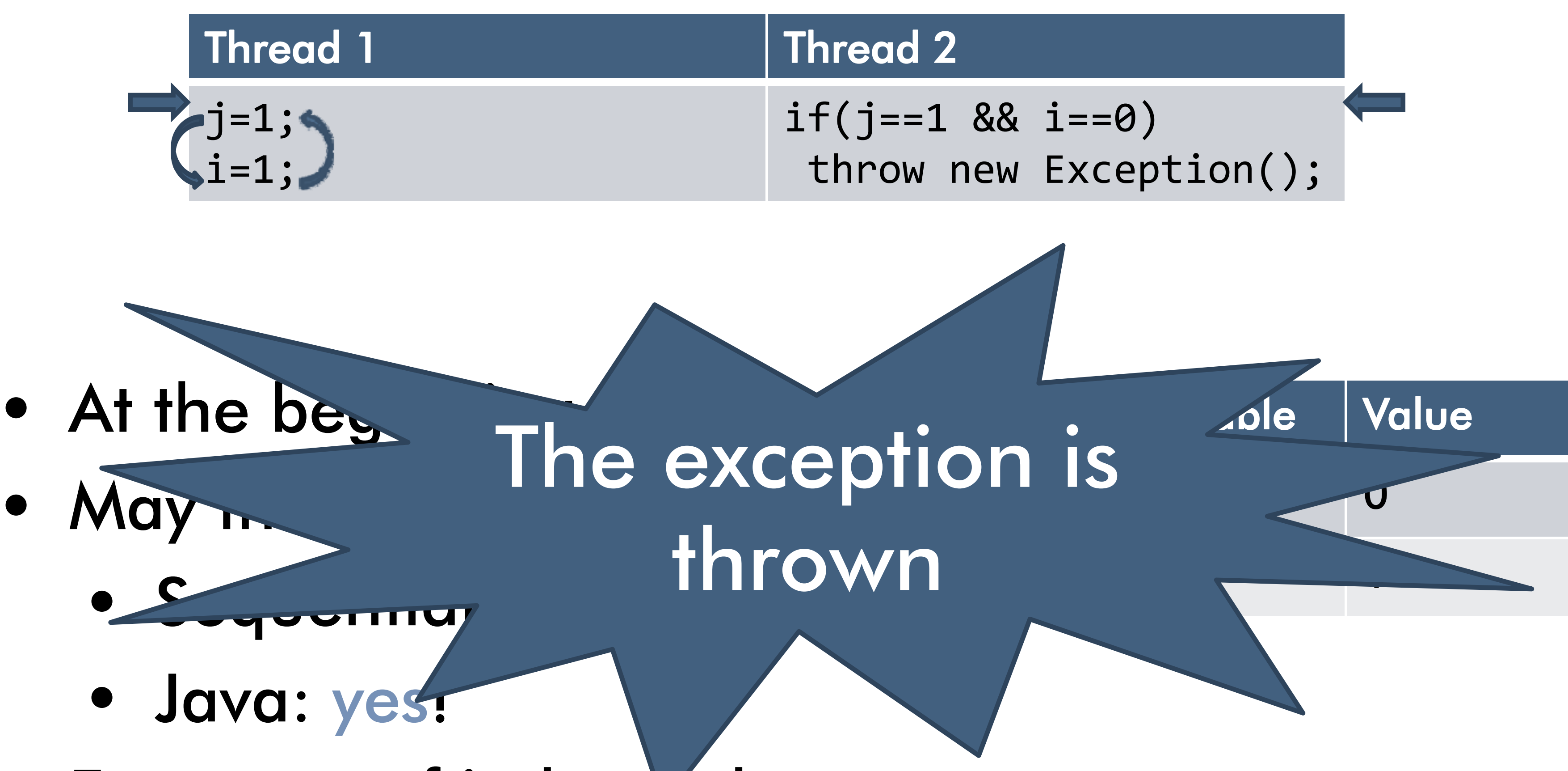
An example

Thread 1	Thread 2
`i=1;` `j=1;`	`if(j==1 && i==0)` `throw new Exception();`

- At the beginning, `i=0`, `j=0`
- May the exception be thrown?
  - Sequential consistency: no!

Variable	Value
`i`	1
`j`	1

An example



Memory models

- Describe how programs interact with memory
  - > Also in presence of data races
- Interesting only for multithreading
  - > Single thread: *"as if sequential"* semantics
- Define which behaviors are allowed
  - > Which parallel writes a read action can see
- Two opposite needs
  - > Make sure programs run quickly (as much as possible)
    - Allow optimizations
  - > Programmers understand programming languages
    - Restrict optimizations

Memory models

- Sequential consistency
  - > Actions appear as executed exactly in the order of the original program
    - Too much restrictive (presence of data races)
- Happens-before memory model (HBMM)
  - > Happens-before order: transitive closure of
    - Program order
    - Synchronize-with order
- Java memory model
  - > Restriction of the HBMM
  - > Complex formalization

Happens-before memory model

- A read `r[v]` can see a write `w[v]` if and only if
  - > `r[v]` does not happens before `w[v]`
  - > There is no `w'[v]` such that
    - `w[v]` happens before `w'[v]`
    - `w'[v]` happens before `r[v]`

Thread 1	Thread 2

i=0;
i=1;
new Thread2().start();
	int v=i;

- `i=0` happens before `i=1`
- `i=1` happens before `int v=i`

Sound static analysis

- Infer and prove properties
  - > at compile time (static)
  - > respected by all possible executions (sound)
- Abstract interpretation
  - > Mathematical theory developed by P. & R. Cousot
  - > Define concrete semantics of programs
  - > Soundly approximate it
- Common approach:
  - > Semantics defined in a fixpoint form

Contribution

- Static analysis sound with respect to MM
  - > Based on abstract interpretation
  - > Instantiated on the HBMM
- Concrete semantics: formalization of the MM
  - > Trace semantics
  - > In fixpoint form
- Abstract semantics: generic with respect to
  - > Property of interest
  - > Numerical domain
- Implemented

Outline

1. Introduction
2. Concrete semantics
3. Abstract semantics
4. Experimental results
5. Conclusion and future work

Formalizing executions

- Trace semantics:
  - > Execution represented as an ordered trace of states
- Common approach in abstract interpretation

```
i=1;
j=1;
```

Var.	Val.
i	0
j	0

→

Var.	Val.
i	1
j	0

→

Var.	Val.
i	1
j	1

- It relies on the *"as if sequential"* semantics
- Set of states: Σ
- Finite traces of states: $\Sigma^{\vec{+}}$

Multithread and multicore

- Problem with multithreading:
  - > Several traces of execution
- Several architectures: single, dual, … cores CPU
- Example: single core

Thread 1	Thread 2
`i=1;` `j=1;`	`if(j==1 && i==0)` `throw new Exception();`

Var.	Val.
i	0
j	0

Th.1 →

Var.	Val.
i	1
j	0

Th.1 →

Var.	Val.
i	1
j	1

Th.2 →

Var.	Val.
i	1
j	1

Dual core

C.1 → i=1;
C.2 → j=1;

Thread 1	Thread 2
`i=1;` `j=1;`	`if(j==1 && i==0)` ` throw new Exception();`

Core 1:

Var.	Val.
i	0
j	0

Th.1 →

Var.	Val.
i	1
j	0

Th.2 →

Var.	Val.
i	1
j	0

Core 2:

Var.	Val.
i	1
j	0

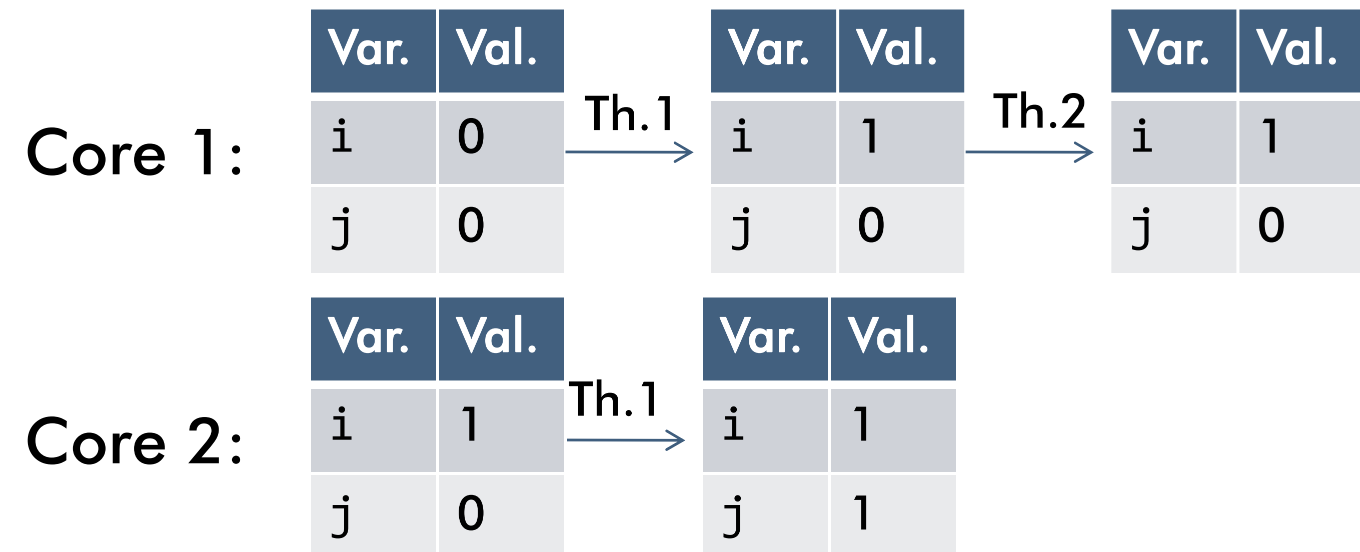
Th.1 →

Var.	Val.
i	1
j	1

Thread partitioning

- Main issue:
  - > Different ways of executing a multithreaded program
  - > Execution relies on the hardware architecture
- Multithreaded model
  - > Several threads are executed in parallel
  - > *"As if sequential"* semantics for each thread locally
  - > Shared memory
- Thread partitioning domain
  - > It collects the trace of execution for each thread
  - > Abstract away the interleaving of different threads

Thread partitioning domain



Thread 1:

Thread 2:

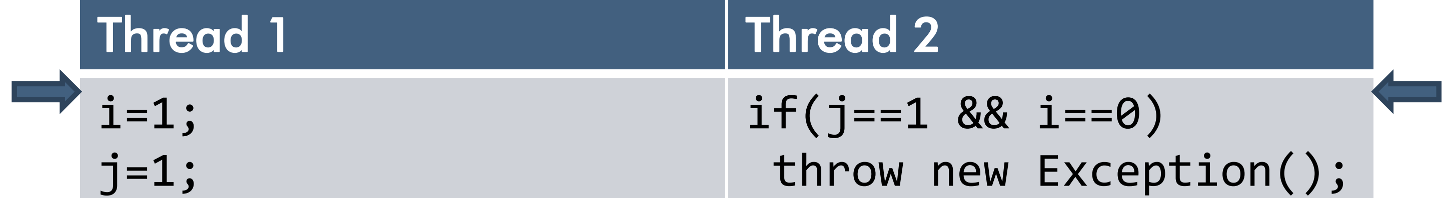
Thread partitioning domain

- We collect for each thread its trace of execution

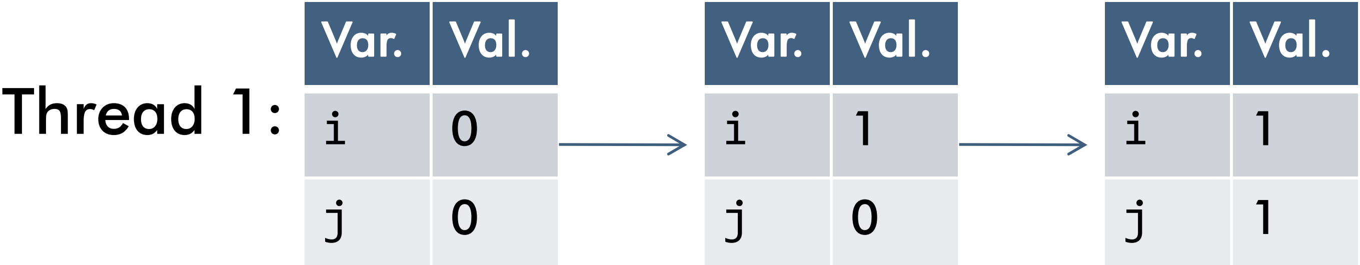
$$\Psi : [\mathrm{Tid} \rightarrow \vec{\Sigma^{+}}]$$

- What is the state of a single thread?
  > Private memory
    - E.g. local variables
  > Shared memory
    - E.g. the heap
- Memory model
  > Define the possible views of the shared memory
- Concrete semantics
  > Choose one of the possible views of the memory

Thread partitioning domain



Thread 1	Thread 2
`i=1;` `j=1;`	`if(j==1 && i==0)` `throw new Exception();`

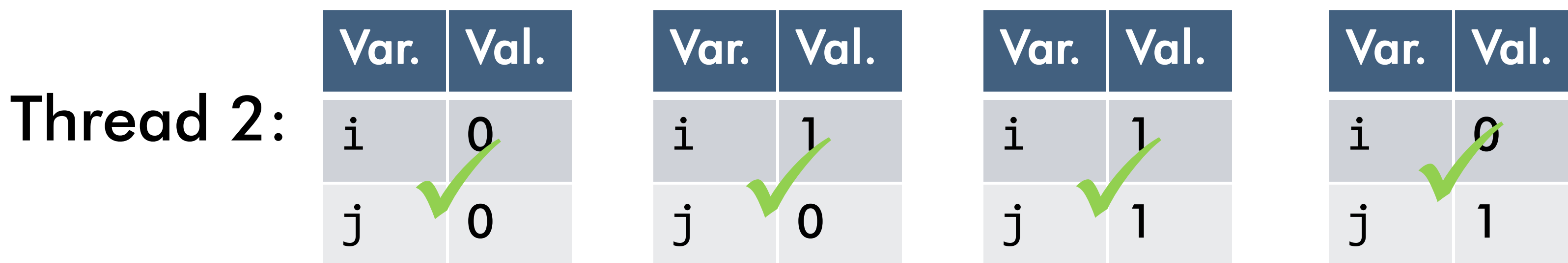


Thread 1:

Var.	Val.
i	0
j	0

Var.	Val.
i	1
j	0

Var.	Val.
i	1
j	1



Thread 2:

Var.	Val.
i	0
j	0

Var.	Val.
i	1
j	0

Var.	Val.
i	1
j	1

Var.	Val.
i	0
j	1

HBMM defines the allowed views of shared memory

Partial finite trace semantics

- Build up all the partial finite executions of a program
- Fixpoint semantics of thread t

$$\mathbb{S}^{\circ}[\![\mathsf{t}]\!] = lfp_{\emptyset}^{\subseteq} \lambda \mathsf{T}.\{\sigma_0^{\mathsf{t}}\} \cup \{\sigma_0 \rightarrow \cdots \rightarrow \sigma_{\mathsf{n}} \rightarrow \sigma_{\mathsf{n}+1} : \sigma_0 \rightarrow \cdots \rightarrow \sigma_{\mathsf{n}} \in \mathsf{T} \wedge \sigma_{\mathsf{n}+1} \in \mathbb{S}[\![\sigma_{\mathsf{n}}]\!]\}$$

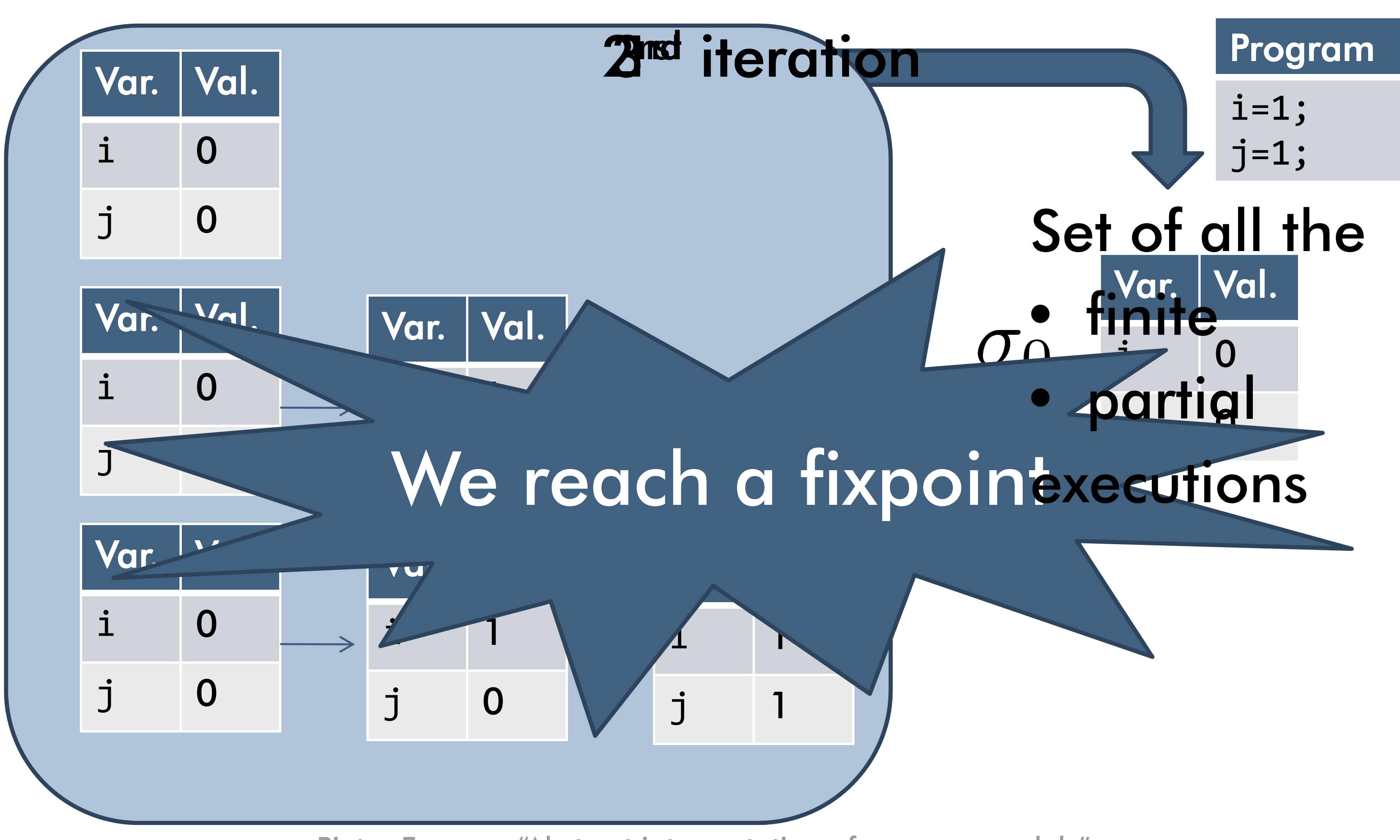
where

> σ_0^{t} is the initial state of thread t

> $\mathbb{S}$ is the small step semantics of the programming language

> $\sigma_0 \rightarrow \cdots \rightarrow \sigma_{\mathsf{n}}$ is a trace composed by n states

Partial finite trace semantics



Something is missing

- Definition of single thread semantics

$$\mathbb{S}^{\circ}[\![\mathsf{t}]\!] = lfp^{\subseteq}_{\emptyset} \lambda \mathsf{T}.\{\sigma^{\mathsf{t}}_{0}\} \cup \{\sigma_0 \rightarrow \cdots \rightarrow \sigma_{\mathsf{n}} \rightarrow \sigma_{\mathsf{n+1}} : \sigma_0 \rightarrow \cdots \rightarrow \sigma_{\mathsf{n}} \in \mathsf{T} \wedge \sigma_{\mathsf{n+1}} \in \mathbb{S}[\![\sigma_{\mathsf{n}}]\!]\}$$

where

> σ_0 is the initi... ate

> $\mathbb{S}$ is...

> ... n states

It should take into account the MM

Small step semantics

- We suppose to have the semantics of statements
  - > *"As if sequential"* semantics
  - > Atomic at multithreaded level
- If we have a thread local action
  - > Follow the semantics of statements
- If we read something from the shared memory
  - > MM provides all possible views of the shared memory
- Non deterministic choice
  - > Different threads' interleaving may cause to read different values from the shared memory

Single-thread semantics

$$\mathbb{S}^{\circ}[\![\mathsf{f}, \mathsf{t}]\!] = \mathit{lfp}_{\emptyset}^{\subseteq} \lambda \mathsf{T}. \{\sigma_0^{\mathsf{t}}\} \cup \{\sigma_0 \rightarrow \cdots \rightarrow \sigma_{\mathsf{n}} \rightarrow \sigma_{\mathsf{n+1}} : \sigma_0 \rightarrow \cdots \rightarrow \sigma_{\mathsf{n}} \in \mathsf{T} \wedge \sigma_{\mathsf{n+1}} \in \mathbb{S}[\![\mathsf{f}, \sigma_{\mathsf{n}}]\!]\}$$

The semantics extracts
the values written in
parallel by other threads

Multithread semantics

- $\mathbb{S}^{\circ}$ exposes values on the shared memory
  - > These values may be read by other threads
  - > They may cause new behaviors
    - Exposing eventually new values on the shared memory
- Multithread semantics
  - > Iterate the single thread semantics on each thread
  - > Each iteration may produce new values on the heap
  - > Stop when a fixpoint is reached
- Not only values on the shared memory
  - > New executions may create and launch new threads
    - Dynamic allocation of threads

Multithread semantics

$$\mathbb{S}^{\parallel}[\![\mathsf{f}_0]\!] = lfp_{\emptyset}^{\subseteq} \lambda\Phi.\{\mathsf{f}_0\} \cup \{\mathsf{f}_i : \exists \mathsf{f}_{i-1} \in \Phi : \forall t \in dom(\mathsf{f}_{i-1}) : \mathsf{f}_i(t) \in \mathbb{S}^{\circ}[\![\mathsf{f}_{i-1}, t]\!] \cap \Sigma_{\mathbb{S}^{\circ}}^{\vec{+}}\}$$

where

- $\Sigma_{\mathbb{S}^{\circ}}^{\vec{+}}$ contains the traces ending with a final state
- f_0 is the function relating
  - > each active thread at the beginning of the execution
  - > to a trace containing its initial state

Multithread semantics in action

Thread 1	Thread 2
`i=1;` `j=1;`	`if(j==1 && i==0)` `throw new Exception();`

Initial state

Th.1:

i	0
j	0

Th.2:

i	0
j	0

1st iteration

Th.1:

i	0
j	0

→

i	1
j	0

→

i	1
j	1

Th.2:

i	0
j	0

→

i	0
j	0

Multithread semantics in action

Thread 1	Thread 2
`i=1;` `j=1;`	`if(j==1 && i==0)` `throw new Exception();`

Initial state ∪ 1st iteration

2nd iteration

Th.1:

i	0
j	0

Th.2:

i	0
j	0

Th.1: i 0, j 0 → i 1, j 0

Th.2: i 0, j 0 → i 0, j 0

Th.1: i 0, j 0 → i 1, j 0 → i 1, j 1

Th.2: i 0, j 0 | i 1, j 0 | i 1, j 1 | i 0, j 1 → i 0, j 0 | i 1, j 0 | i 1, j 1

Thread 1 does not read from the shared memory!

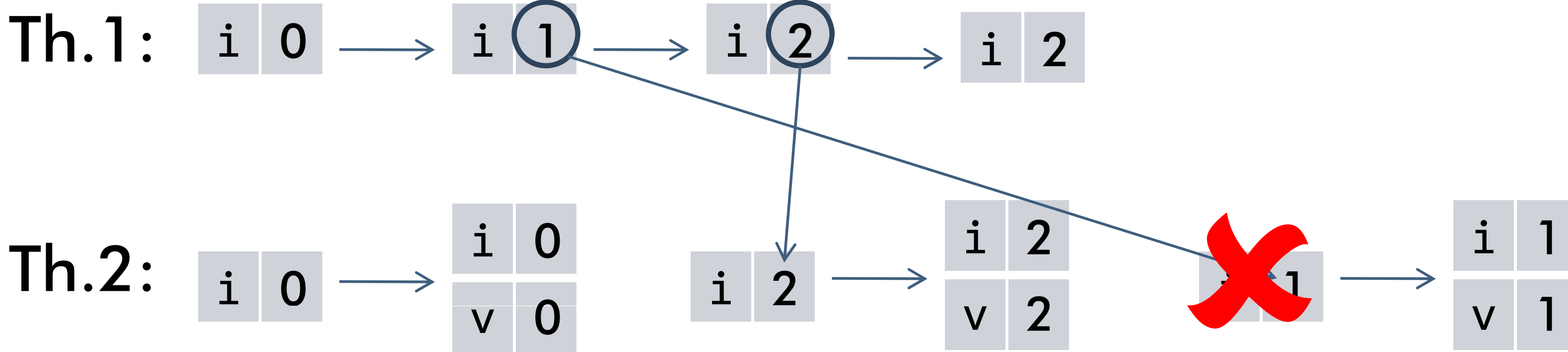
Exception

Synchronization patterns

- Our formalization supports
  - > Monitors

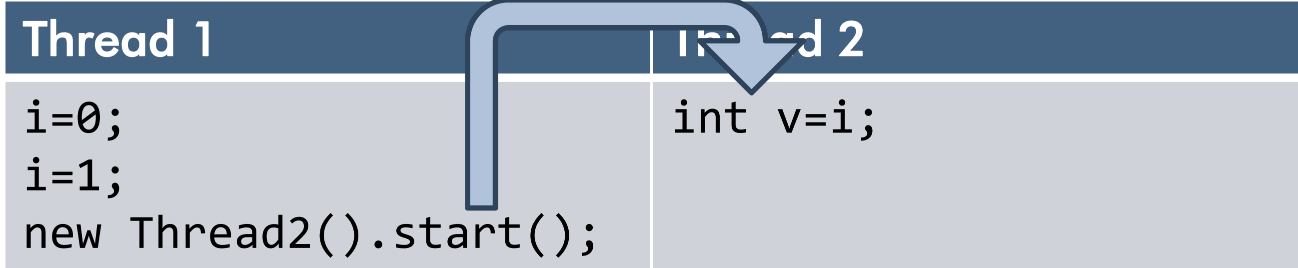
Thread 1	Thread 2

lock(o);
i=1;
i=2;
unlock(o);
	lock(o);
	int v=i;
	unlock(o);



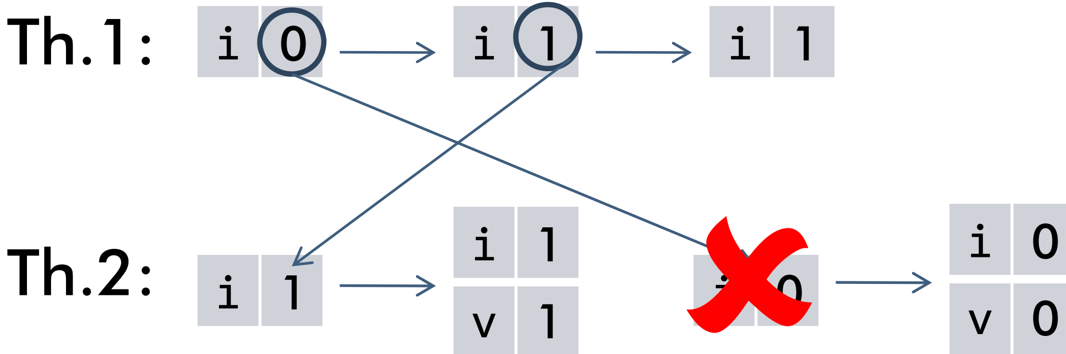
Synchronization patterns

- Our formalization supports
  - Dynamic launch of threads



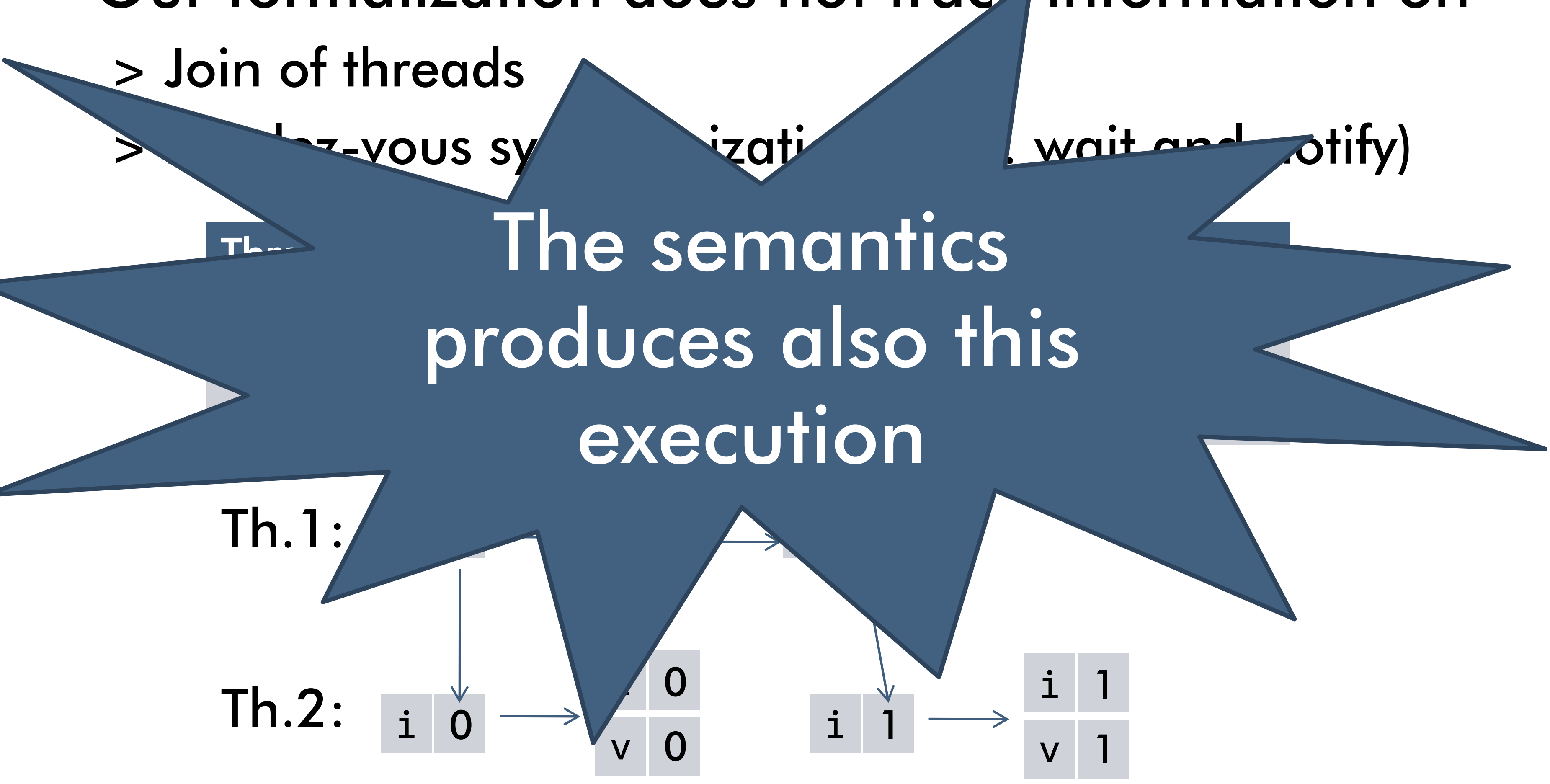
Thread 1	Thread 2

`i=0;`
`i=1;`
`new Thread2().start();`
	`int v=i;`



Synchronization patterns

- Our formalization does not track information on



Soundness

- Final goal:
  > Formalize a static analysis
  > Sound with respect to all the behaviors of a MM
- We do not consider some synchronization actions
  > Then we allow more behaviors
- This does not break the soundness
  > … but it will produce less precise results
- Future work
  > Track information on other synchronization patterns

Outline

1. Introduction
2. Concrete semantics
3. Abstract semantics
4. Experimental results
5. Conclusion and future work

Abstract semantics

- Concrete semantics
  - Not computable
- Abstract semantics
  - Approximation of the concrete semantics
  - Computable
  - Soundness proved formally
- One abstract trace approximates all concrete traces
  - Common approach in abstract interpretation
  - One abstract state for each statement in the program

Abstract domain

$$\overline{\Psi} : [\overline{\text{Tid}} \rightarrow \overline{\Sigma}^{\vec{+}}]$$

- What is an abstract state?
  > We do not define it in the details
  > We interact with it using some functions
- Generic analysis
  > E.g., it can be instantiated with different numerical domains
  - Intervals, signs, …
  - Up to know, only non-relational numerical domains

Single thread abstract semantics

$$\overline{\mathbb{S}}^{\circ}[\![\overline{\mathsf{f}},\overline{\mathsf{t}}]\!] = lfp_{\emptyset}^{\sqsubseteq^{\vec{+}}} \lambda\overline{\tau}.\{\overline{\sigma_0^{\mathsf{t}}}\} \sqcup^{\vec{+}} \{\overline{\sigma_0} \rightarrow \cdots \rightarrow \overline{\sigma_n} \rightarrow \overline{\sigma_{n+1}} : \overline{\tau} = \overline{\sigma_0} \rightarrow \cdots \rightarrow \overline{\sigma_n} \wedge \overline{\sigma_{n+1}} = \overline{\mathbb{S}}[\![\overline{\mathsf{f}},\overline{\sigma_n}]\!]\}$$

where

- $\sqcup^{\vec{+}}$ and $\sqsubseteq^{\vec{+}}$ are respectively the applications of the upper bound and the partial order operator of states to all the elements of the traces

$$\mathbb{S}^{\circ}[\![\mathsf{f},\mathsf{t}]\!] = lfp_{\emptyset}^{\subseteq} \lambda\mathsf{T}.\{\sigma_0^{\mathsf{t}}\} \cup \{\sigma_0 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma_{n+1} : \sigma_0 \rightarrow \cdots \rightarrow \sigma_n \in \mathsf{T} \wedge \sigma_{n+1} \in \mathbb{S}[\![\mathsf{f},\sigma_n]\!]\}$$

Multithread semantics

$$\overline{\mathbb{S}^{\|}}[\![\overline{f_0}]\!] = lfp_{\emptyset}^{\overline{\sqsubseteq}^{\Phi}} \lambda \overline{f}.\{\overline{f_0}\} \sqcup^{\Phi} \{\overline{f_i} : \forall \overline{t} \in dom(\overline{f}) :$$

$$\overline{f_i}(\overline{t}) = \overline{\mathbb{S}^{\circ}}[\![\overline{f}, \overline{t}]\!] \sqcap^{\Phi} \overline{\Sigma^{\vec{+}}_{\mathbb{S}^{\circ}}}\}$$

where $\sqcup^{\Phi}, \sqcap^{\Phi}$ and $\overline{\sqsubseteq}^{\Phi}$ are respectively the applications of $\sqcup^{\vec{+}}, \sqcap^{\vec{+}}$ and $\overline{\sqsubseteq}^{\vec{+}}$ to the traces of all the threads

$$\mathbb{S}^{\|}[\![f_0]\!] = lfp_{\emptyset}^{\subseteq} \lambda \Phi.\{f_0\} \cup$$

$$\{f_i : \exists f_{i-1} \in \Phi : \forall t \in dom(f_{i-1}) :$$

$$f_i(t) \in \mathbb{S}^{\circ}[\![f_{i-1}, t]\!] \cap \Sigma^{\vec{+}}_{\mathbb{S}^{\circ}}\}$$

Abstract semantics in action

Thread 1	Thread 2
`i=1;` `j=1;`	`if(j==1 && i==0)` `throw new Exception();`

Initial state

Th.1:

i	[0..0]
j	[0..0]

Th.2:

i	[0..0]
j	[0..0]

1st iteration

Th.1:

i	[0..0]
j	[0..0]

→

i	[1..1]
j	[0..0]

→

i	[1..1]
j	[1..1]

Th.2:

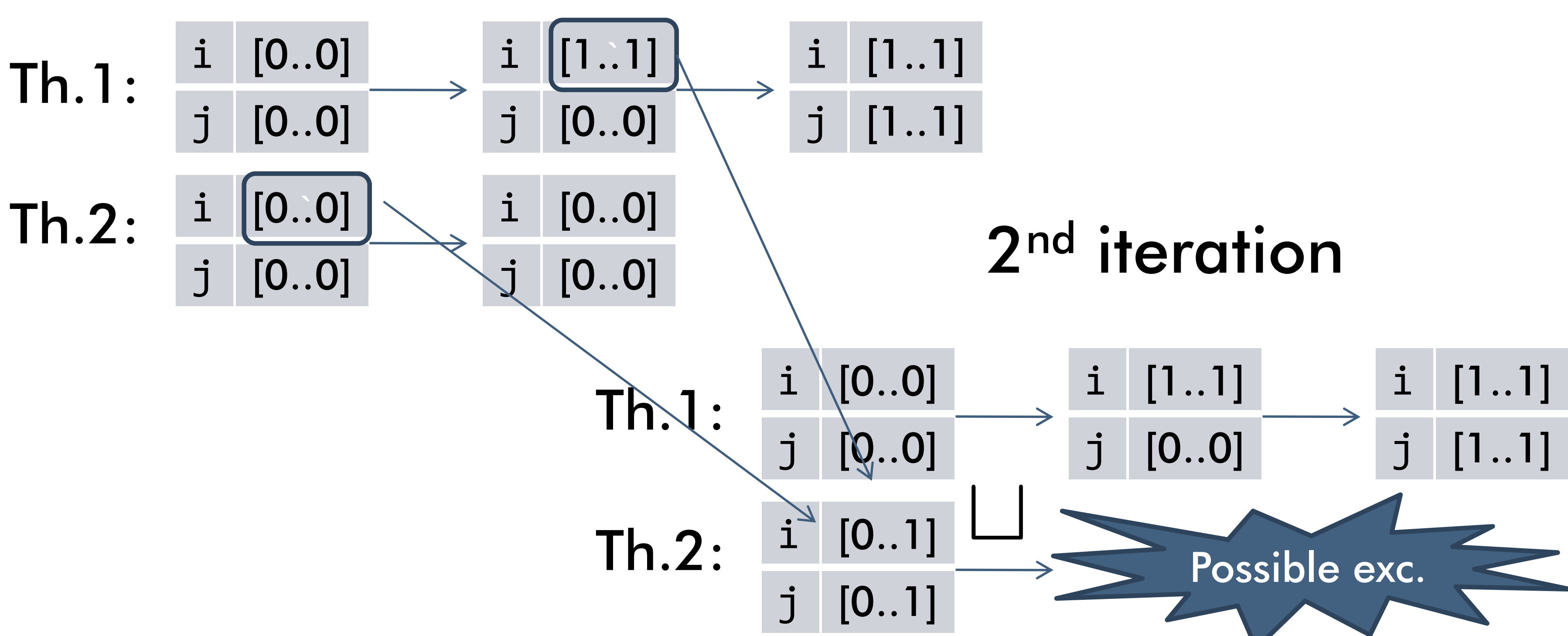
i	[0..0]
j	[0..0]

→

i	[0..0]
j	[0..0]

Multithread semantics in action

Thread 1	Thread 2
`i=1;` `j=1;`	`if(j==1 && i==0)` ` throw new Exception();`



Abstract threads

- We support dynamic allocation of threads
  > Unbounded number of concrete threads
- Finite approximation
  > One abstract thread may represent many concrete threads
  > For instance, Java
    - Threads are objects
    - Threads are identified by reference
    - One abstract reference may represent many concrete references
- Self interleaving of abstract threads

An example

Main Thread	Thread

`i=0;`
`for(int j=0;j<input;j++)`
`        new Thread().start();`
	`i++;`

MainThread: j 0 → j 1 → j 2 → j 3 →

Thread: i 0 → i 1

Thread: i 1 → i 2

Thread: i 2 → i 3

Self interleaving

Main Thread	Thread

`i=0;`
`for(int j=0;j<input;j++)`
`        new Thread().start();`
	`i++;`

MainThread: j [0..0] → j [0..+

Thread: i [0..0] ⊔ → i [1..

Thread: i [0..1] → i [1..

Thread: i [0..2] → i [1..3]

3 ...ration

After K iterations we apply widening

Thread: i [0..+∞] → i [1..+∞] n[th] iteration

Java

- This framework has been applied to Java
- Everything is identified or accessed by reference
  - > Threads are objects
  - > The shared memory is the heap
  - > Monitors are defined on objects
- We need to develop a specific alias analysis
  - > That is, how we approximate reference
  - > May aliasing
    - Identify threads
    - Discover conflicts on the heap
  - > Must aliasing to check mutual exclusion on monitors

Outline

1. Introduction
2. Concrete semantics
3. Abstract semantics
4. Experimental results
5. Conclusion and future work

Implementation

- Implemented in Checkmate

```
http://www.pietro.ferrara.name/checkmate
```

- Generic analyzer of multithreaded programs
- It analyzes Java bytecode
- Pluggable with different
  - > Numerical non-relational domains
  - > Properties
  - > Memory models
    - HBMM
    - Two relaxed versions

Demo

Eclipse

Case studies

Th1	Th2

`r1=x;`
`y=1;`
`r2=x;`
	`x=1;`
	`r3=y;`

Th1	Th2

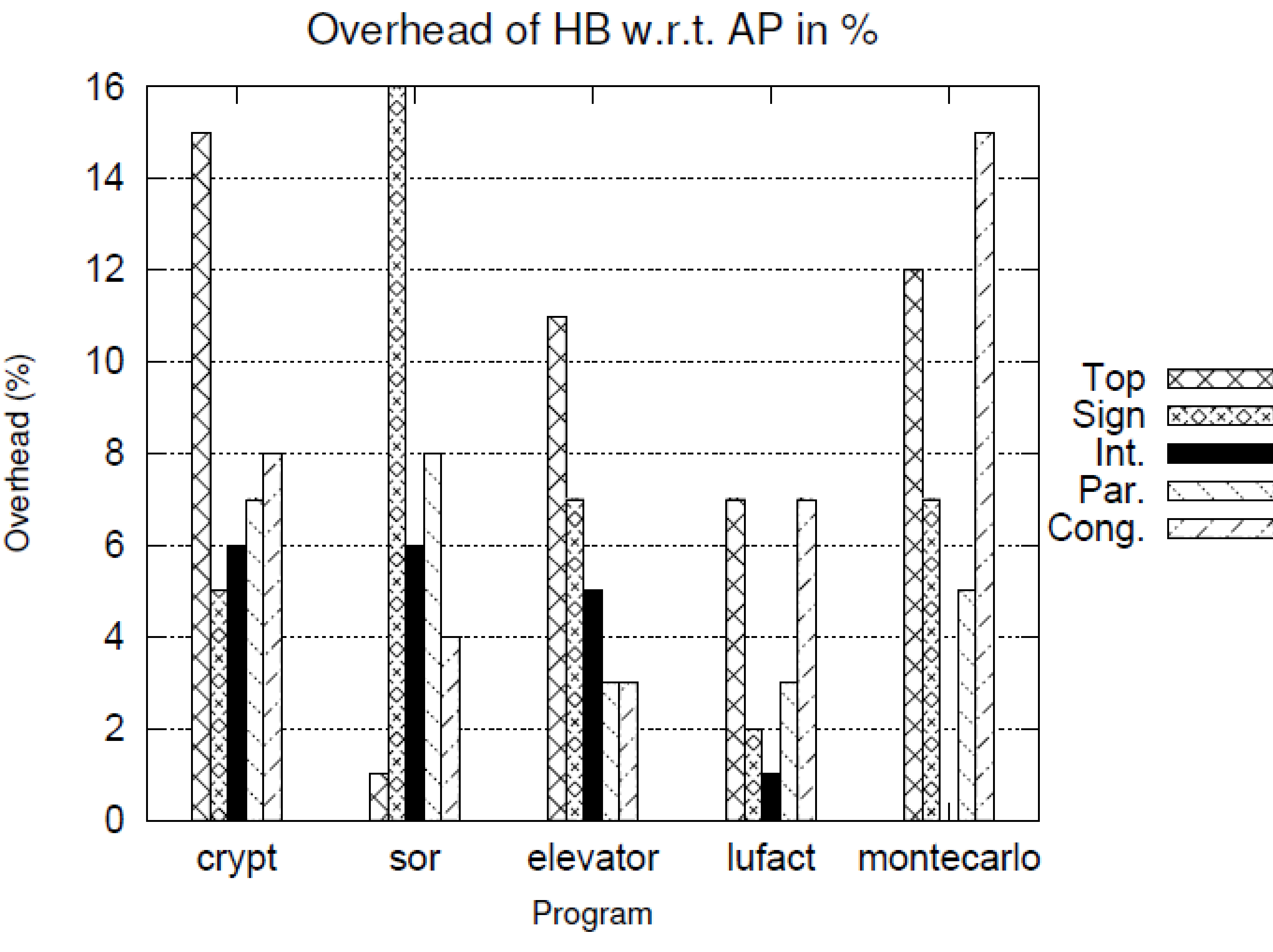
`r1=x;`
`if(r1!=0)`
`  y=42;`
	`r2=y;`
	`if(r2!=0)`
	`  x=42;`

- Applied to eight case studies taken from JMM
- Usually we catch the behaviors precisely
- Too much approximated in some cases
  - > In particular when dealing with volatile variables
  - > Synchronization not taken into account
    - We may extend our formalization

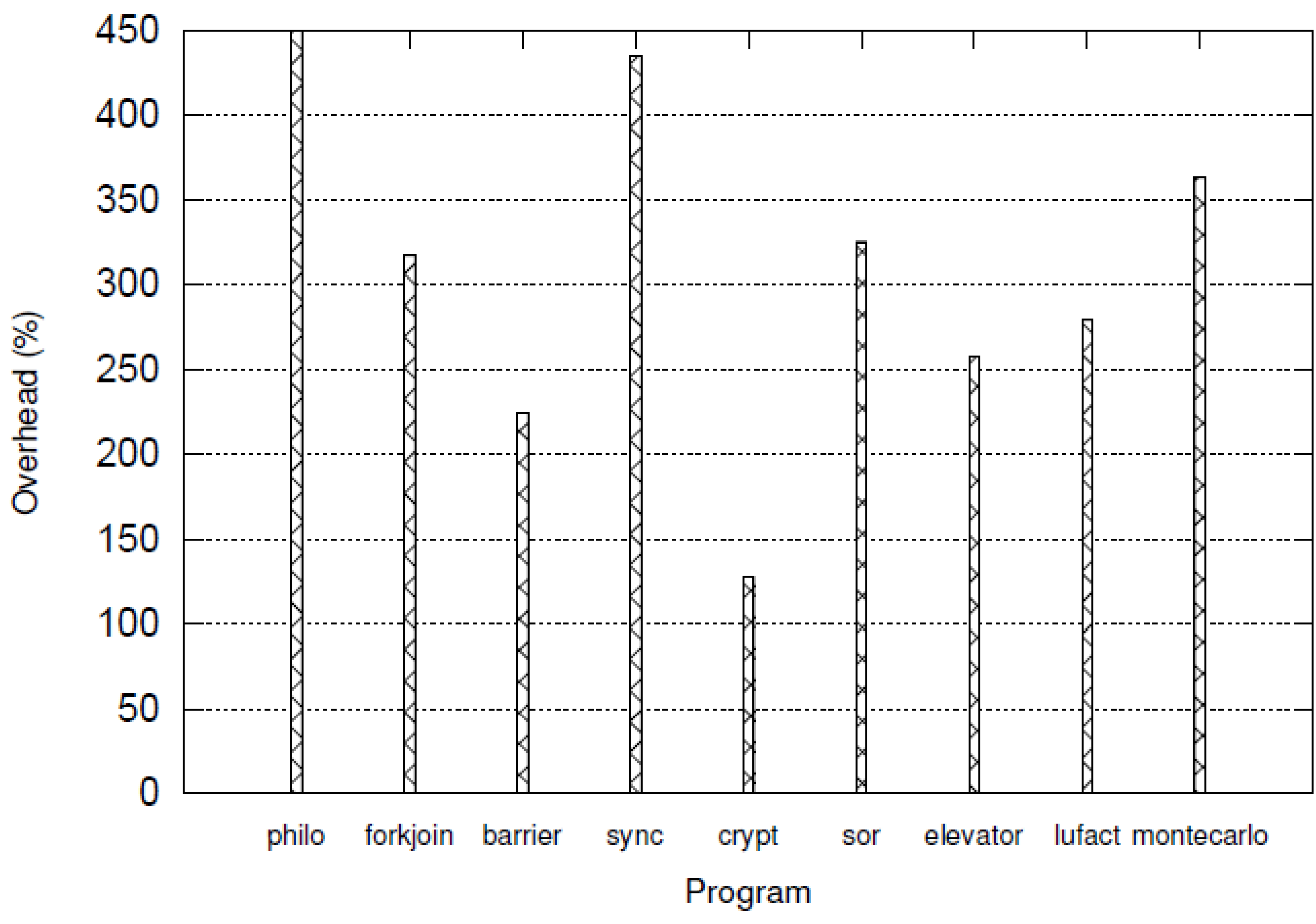
Benchmarks

- An incremental application
  - > HBMM compared to two more approximated MMs
  - > Study the overhead of more precise MMs
  - > Overhead of HBMM increases linearly
- Some well-known benchmarks
  - > We are in position to analyze programs composed by
    - An unbounded number of threads
    - Some thousands of bytecode statements
  - > Usually in a couple of minutes

Overhead of HBMM



Overhead multithread semantics



Outline

1. Introduction
2. Concrete semantics
3. Abstract semantics
4. Experimental results
5. Conclusion and future work

Conclusion

- We proposed a
  - > Generic static analysis
  - > Of multithreaded programs
  - > Sound with respect to HBMM
- The framework is more generic
  - > It could be applied to other MM
  - > It can be extended to other synchronization patterns
  - > It can be applied to different programming languages
- Applied to Java bytecode multithreaded programs
  - > Fast and precise

Limits

- The overall analysis does not scale up
- Support only few synchronization patterns
  - > Monitors
  - > Launch of threads
- We do not support
  - > Join of threads
  - > `volatile` variables
  - > Rendez-vous style synchronizations
    - Difficult to track them precisely and efficiently

Future work - Precision

- Support more synchronization patterns
  - > In particular, rendez-vous style patterns
- Support relational numerical domains
  - > Up to now, only non-relational domains
    - Sign, Intervals, …
  - > Relational information required by industrial programs
- Apply the framework to other memory models
  - > Java MM
  - > Sequential consistency

Future work - Scalability

- Usually, few static analyzers scale up
  - > Necessary modular reasoning
  - > For instance, relying on contracts
    - Clousot @ Microsoft
  - > Checkmate: 1st generic analyzer of multithreading
- Contracts to modularly reason on multithreading
  - > For instance, Chalice
- Idea: modular analysis based on these contracts

Thank you!