

Static analysis via abstract interpretation of multithreaded programs

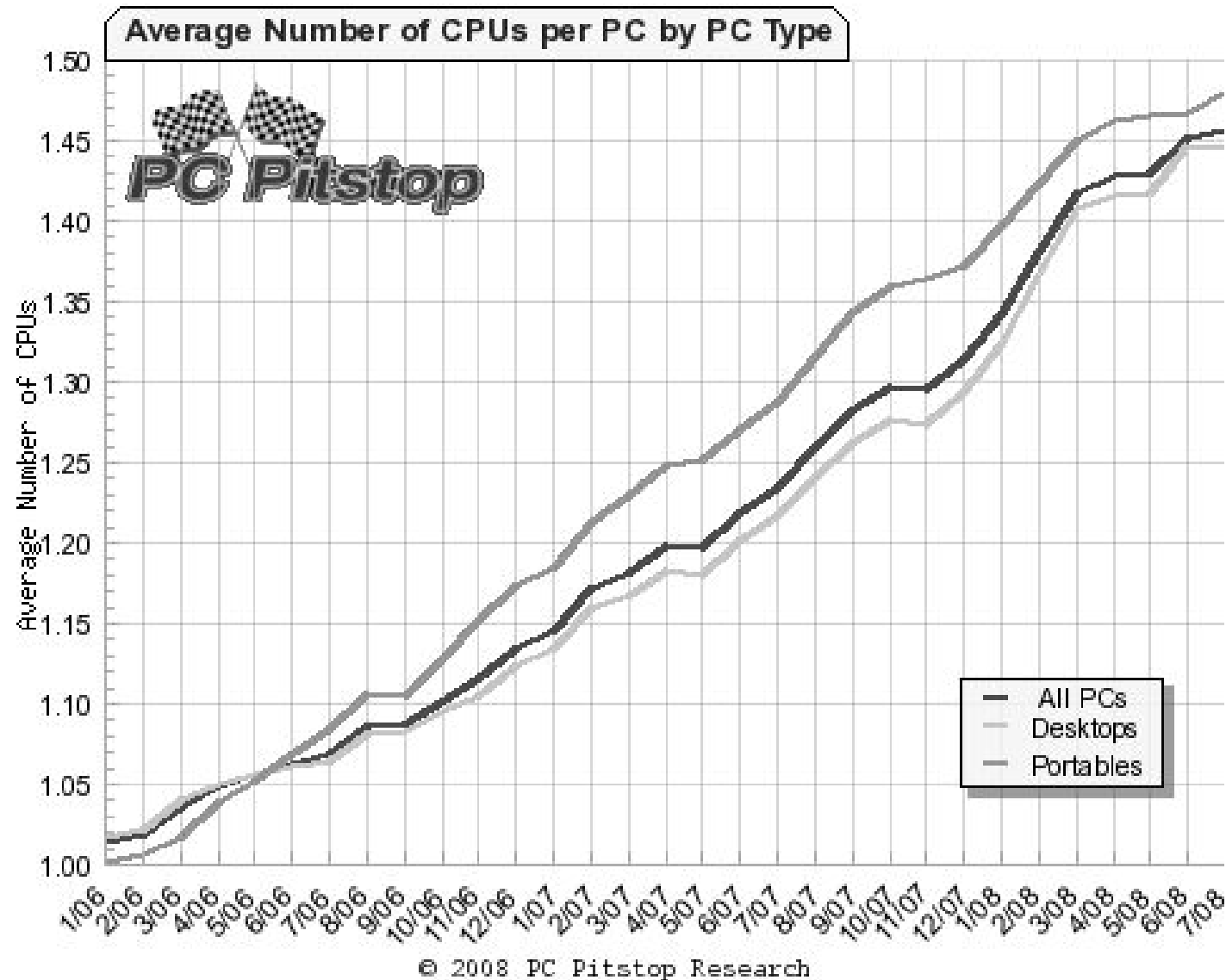
Pietro Ferrara

Chair of Programming Methodology
ETH Zurich
Switzerland

August 25, 2009

IRISA-INRIA, Rennes, France

Multicore revolution



Multicore revolution

- The only way to try to prolong **Moore's law**
- Today: at least dual core processors
- Current trend: **manycore**
 - > Quad cores: 150 € (AMD Phenom X4 9650)
 - > Eight cores: server processors (e.g. AMD Opteron)
 - > Sixteen cores: soon...
- Sequential programs do not exploit multicores
- Applications with **explicit parallelism**

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
 - > C#
- Implicit communications via shared memory
- Synchronization on monitors
- **Subtle** and problematic

Motivating Examples

ThreadIncrease	Main Thread
<code>a.i++;</code>	<code>a.i=0; for(int j=0; j<N; j++) new ThreadIncrease(a).start(); if(a.i>1000) throw new Exception();</code>

- In order to expose the exception, we need that
 - > $N > 1000$
 - > Main thread reads `a.i` after at least **1.000 threads** read and increased it
- Really particular execution
 - > Exception **rarely exposed** by testing
 - > Difficult to reproduce this execution

Motivating Examples

Deposit 1	Deposit 2
<pre>int t1=a.amount; t1=t1+1000\$; a.amount=t1;</pre>	<pre>int t2=a.amount; t2=t2+1000\$; a.amount=t2;</pre>

- Object a shared between both threads
- Field amount declared as volatile
 - > All accesses are synchronized
- No data race
 - > Writes and reads are synchronized
- Nondeterministic behavior
 - > 1.000\$ may “disappear”
 - > Particular interleaving of threads' executions

Motivating Examples

Thread 1	Thread 2
<pre>a.i=1; a.j=1;</pre>	<pre>if(a.j==1 && a.i==0) throw new Exception();</pre>

- At the beginning:
 - > $i=0$
 - > $j=0$
- The exception may be **thrown**
 - > Thread 2 may see the values written by Thread 1 in a different order
- Memory model
 - > Specify which behaviors are allowed

Static analysis

“Parallel programming is going to require better programming tools to systematically find defects, help debug programs, find performance bottlenecks, and aid in testing. (...) These tools use static program analysis”

Herb Sutter and James Larus. *Software and the concurrency revolution*. In *ACM Queue*. ACM Press, 2005.

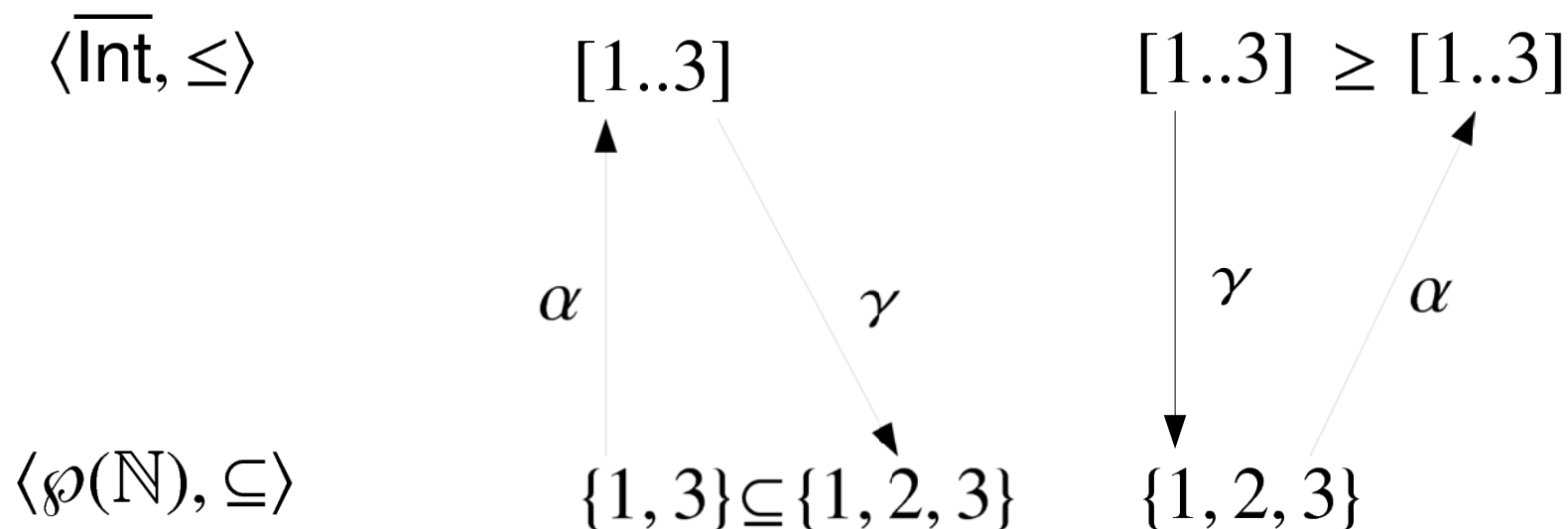
- Testing can expose only **few multithreaded executions**
 - > Some executions exposed only by specific VM
 - > Difficult to reproduce an execution
- Not sufficient to effectively debug multithreading
- Thus static analysis is **appealing** for multithreading
 - > Infer and prove properties at **compile time** satisfied by **all possible executions**

Static analysis

- Tradeoff between precision and efficiency
- Abstract interpretation
 - > Mathematical theory developed by P. & R. Cousot
 - > Define semantics of programs
 - > Soundly approximate it
- Main components:
 - > Semantics of the program
 - > Domain
 - > Property of interest

Domain


- Concrete: runtime behaviors, not computable
- Abstract: computable approximation
- Abstraction and concretization functions
 - > Soundness formally proved
 - Domains: Galois connection
- E.g.: integer values abstracted with intervals



Semantics

- Concrete \mathbb{S} and abstract $\bar{\mathbb{S}}$
- Soundness: $\forall c \in C : \alpha(\mathbb{S}[c]) \leq \bar{\mathbb{S}}[\alpha(c)]$
- E.g., sum using intervals

$$\alpha(\{1, 3\} + \{1, 3\}) = \alpha(\{2, 4, 6\}) = [2..6]$$

$$\alpha(\{1, 3\}) \oplus \alpha(\{1, 3\}) = [1..3] \oplus [1..3] = [2..6]$$


- Usually semantics defined in **fixpoint** form
 - > Small step semantics of statements
 - Locally sound
 - > Fixpoint computes all the executions

Related work

- **Much work on analysis of multithreaded programs:**
 - > **Specific properties**
 - E.g. data races and deadlocks
 - > **Not sound** for all the possible executions
 - E.g. bounded model checking
- **Generic analyzers based on abstract interpretation**
 - > **Successfully applied to sequential programs**
 - > **They can be equipped** with several numerical domains
 - Tradeoff between precision and efficiency
 - > **The same analyzer is instantiated to various properties**
 - > **Sound** w.r.t. all the possible executions

Contribution

- **Generic approach** to the analysis of multithreading
 - > Abstract semantics sound w.r.t. a memory model
- **Formalization of a specific property**
 - > Non-determinism due to threads' interleaving
- **Framework applied to Java bytecode programs**
 - > Alias analysis
 - Identify threads
 - Check accesses on the shared memory → heap
 - Synchronization through monitors → objects
- **Complete implementation**
 - > Checkmate
 - Generic static analyzer of Java multithreading

Outline

1. Introduction

2. Happens-before memory model

- Definition in fixpoint form and abstraction

3. Determinism of multithreaded programs

- Formalization of a specific property

4. Domain and semantics of Java bytecode

- Low-level domain, specific alias analysis

5. Checkmate

- Generic sound analyzer of multithreaded programs

Memory Model (MM)

- Define which multithreaded behaviors are allowed
 - > Restrict non-determinism
 - > Allow the most part of
 - compiler optimizations
 - existing virtual machines
 - existing processors
- Java MM introduced in 2005, runtime information
- Happens-before MM (HBMM) – L. Lamport 1978
- Main components:
 - > Program order (intra-thread order of statements)
 - > Synchronizes-with relation

HB order and consistency rule

- Happens-before order \rightsquigarrow :
 - > Transitive closure of
 - Program order
 - Synchronizes-with relation
- Core: consistency rule
 - > Specify which values written in parallel are visible
- Happens-before consistency rule:
 - > A read r of a variable v may see a write w to v if:
 - NOT($r \rightsquigarrow w$)
 - There is no w' to v such that $w \rightsquigarrow w' \rightsquigarrow r$
- We focus on
 - > Mutual exclusion
 - > Launch of a thread

Concrete domain and semantics

- **Generic** w.r.t. programming language
- **Collect** for each thread its trace of execution
$$\Psi : \text{TId} \rightarrow \text{St}^{\vec{\tau}}$$
- **Abstract away the inter-thread order of execution**
 - > Consider each thread separately
- **The semantics computes all possible executions**
$$\langle \wp(\Psi), \subseteq, \emptyset, \Psi, \cup, \cap \rangle$$
- *step* function returns the values visible w.r.t. HBMM
- **Intra-thread semantics**
 - > Partial trace semantics
$$\mathbb{S}^\circ : \Psi \times \Omega \times \text{TId} \mapsto \wp(\text{St}^{\vec{\tau}})$$
$$\mathbb{S}^\circ [[f, r, t]] = \text{lfp}_\emptyset^{\subseteq} \lambda T. \{ \sigma_0 \} \cup \{ \sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \rightarrow \sigma_i : \\ \sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \in T \wedge \sigma_i \in \text{step}(t, f, r) \}$$

Concrete semantics

- Multithread semantics

$$\mathcal{S}^{\parallel} : \Psi \times \Omega \mapsto \wp(\Psi \times \Omega)$$

$$\mathcal{S}^{\parallel} \llbracket f_0, r_0 \rrbracket = \text{lfp}_{\emptyset}^{\subseteq} \lambda \Phi. \{(f_0, r_0)\} \cup \{(f_i, r) : \exists (f_{i-1}, r) \in \Phi : \\ \forall t \in \text{dom}(f_{i-1}) : f_i(t) \in \mathcal{S}^{\circ} \llbracket f_{i-1}, r, t \rrbracket, \\ f_i(t) = \sigma_0 \rightarrow \dots \rightarrow \sigma_i, \sigma_i \in \text{St}_{\rightarrow}^{\circ}\}$$

- Two nested fixpoints

- > Each iteration of the multithread semantics:

- Produces **new multithreaded executions**

- > That may expose **new values** on shared memory

- > That may produce **new executions** of other threads

- > ...

The Example

Thread 1	Thread 2
<code>a.i=1;</code> <code>a.j=1;</code>	<code>if(a.j==1 && a.i==0)</code> <code>throw new Exception();</code>

- 1st iteration of the multithread semantics

> *visible* : `i=0`
`j=0`

Thread 1: `i=0` → `i=1` → `i=1`
`j=0` → `j=0` → `j=1`

Thread 2: `i=0` → `j==1 && i==0 ?`
`j=0` → `false`

The Example

Thread 1	Thread 2
<code>a.i=1;</code> <code>a.j=1;</code>	<code>if(a.j==1 && a.i==0)</code> <code>throw new Exception();</code>

- 2nd iteration of the multithread semantics

> *visible* :

<code>i=0</code>	<code>i=1</code>	<code>i=0</code>	<code>i=1</code>
<code>j=0</code>	<code>j=0</code>	<code>j=1</code>	<code>j=1</code>

Thread 1: `i=0` → `i=1` → `i=1`
`j=0` → `j=0` → `j=1`

Thread 2: `i=1` → `j==1 && i==0 ?`
`j=0` → `false`

The Example

Thread 1	Thread 2
<pre>a.i=1; a.j=1;</pre>	<pre>if(a.j==1 && a.i==0) throw new Exception();</pre>

- 2nd iteration of the multithread semantics

> *visible* :

i=0	i=1	i=0	i=1
j=0	j=0	j=1	j=1

Thread 1: i=0 → i=1 → i=1
 j=0 → j=0 → j=1

Thread 2: i=0 → j==1 && i==0 ?
 j=1 true

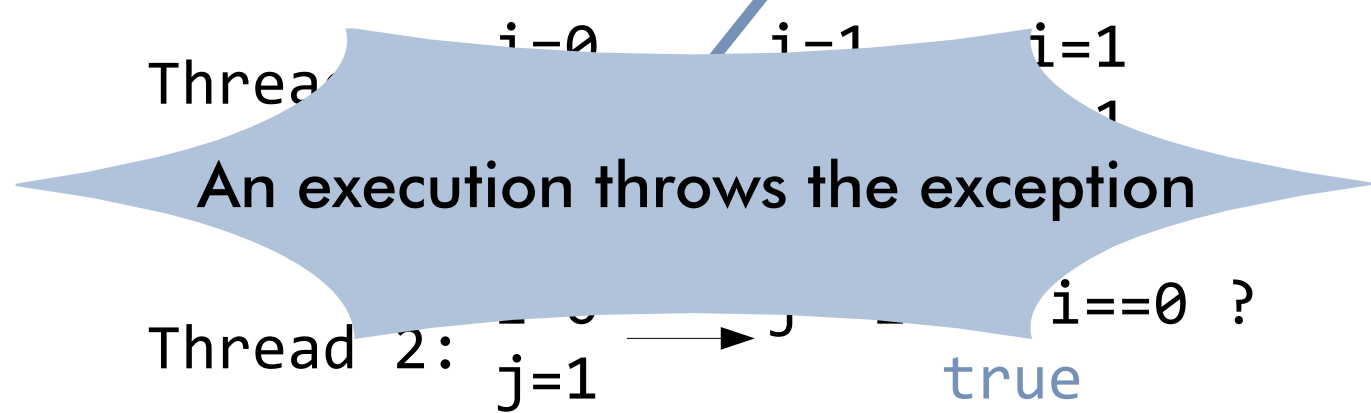
The Example

Thread 1	Thread 2
<pre>a.i=1; a.j=1;</pre>	<pre>if(a.j==1 && a.i==0) throw new Exception();</pre>

- 2nd iteration of the multithread semantics

> *visible* :

i=0	i=1	i=0	i=1
j=0	j=0	j=1	j=1



Abstract domain and semantics

- Pointwise abstraction of the concrete definitions:

$$\bar{\Psi} : \text{TId} \rightarrow \overline{\text{St}}^{\vec{+}}$$

- One trace abstracts all the executions
 - > Upper bound of all the visible values

$$\bar{\mathbb{S}}^{\circ} : [(\bar{\Psi} \times \bar{\Omega} \times \text{TId}) \rightarrow \overline{\text{St}}^{\vec{+}}]$$

$$\bar{\mathbb{S}}^{\circ} [\bar{f}, \bar{r}, t] = \text{lfp}_{\epsilon}^{\sqsubseteq_{\tau}} \lambda \bar{\tau}. \{\bar{\sigma}_0\} \sqcup_{\tau} \{\bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_{i-1} \rightarrow \bar{\sigma}_i : \bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_{i-1} = \bar{\tau} \wedge \bar{\sigma}_i = \overline{\text{step}}(t, \bar{f}, \bar{r}, \bar{\sigma}_{i-1})\}$$

$$\bar{\mathbb{S}}^{\parallel} : [\bar{\Psi} \times \bar{\Omega} \rightarrow \bar{\Psi} \times \bar{\Omega}]$$

$$\bar{\mathbb{S}}^{\parallel} [\bar{f}_0, \bar{r}_0] = \text{lfp}_{\emptyset}^{\sqsubseteq_f} \lambda (\bar{f}, \bar{r}). \{(\bar{f}_0, \bar{r}_0)\} \sqcup_f \{(\bar{f}_i, \bar{r}) : \forall t \in \text{dom}(\bar{f}) : \bar{f}_i(t) = \bar{\mathbb{S}}^{\circ} [\bar{f}, \bar{r}, t]\}$$

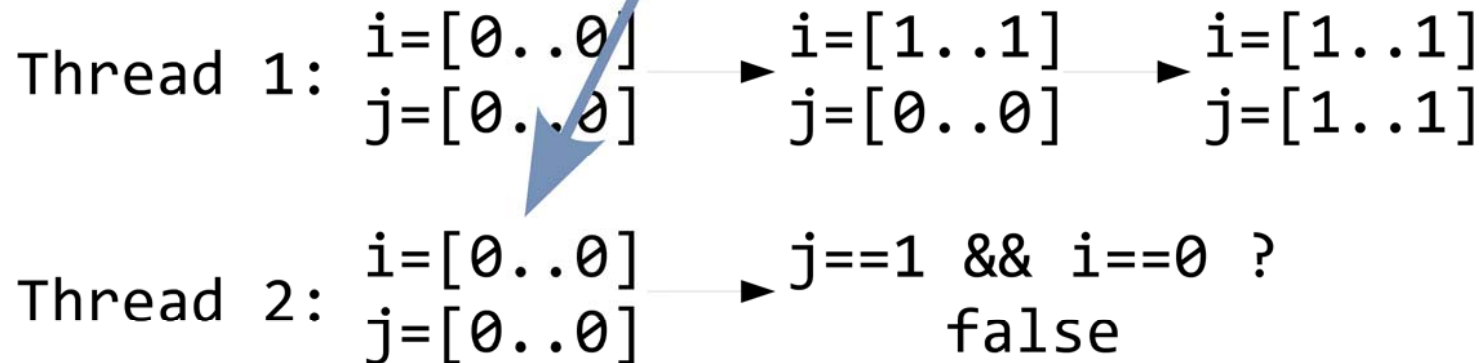
- Sound:** $\forall (\bar{f}, \bar{r}) \in \bar{\Psi}_{pre} \times \bar{\Omega}_{pre} : \alpha_f(\bar{\mathbb{S}}^{\parallel}) [\bar{f}, \bar{r}] \sqsubseteq_f \bar{\mathbb{S}}^{\parallel} [\bar{f}, \bar{r}]$

The Example

Thread 1	Thread 2
<code>a.i=1;</code> <code>a.j=1;</code>	<code>if(a.j==1 && a.i==0)</code> <code>throw new Exception();</code>

- 1st iteration of the multithread semantics

> *visible* : `i=[0..0]`
`j=[0..0]`

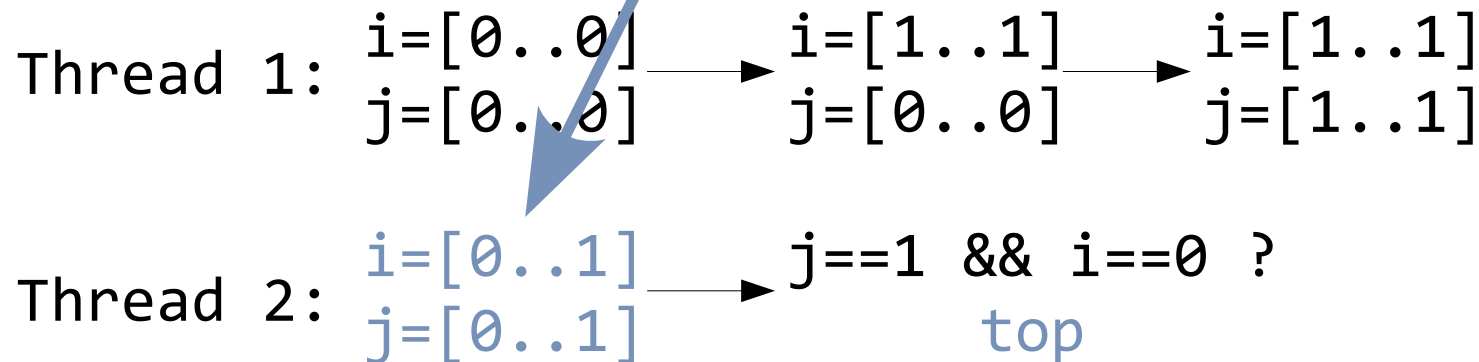


The Example

Thread 1	Thread 2
<code>a.i=1;</code> <code>a.j=1;</code>	<code>if(a.j==1 && a.i==0)</code> <code>throw new Exception();</code>

- 2nd iteration of the multithread semantics

> *visible* : $i=[0..1]$
 $j=[0..1]$



The Example

Thread 1	Thread 2
<code>a.i=1;</code> <code>a.j=1;</code>	<code>if(a.j==1 && a.i==0)</code> <code>throw new Exception();</code>

- 2nd iteration of the multithread semantics

> *visible* : $i=[0..1]$
 $j=[0..1]$

Thread 1: $i=[0..1]$ $j=[0..1]$

The program may throw the exception

Thread 2: $i=[0..1]$ $j=[0..1]$ top

Outline

1. Introduction

2. Happens-before memory model

- Definition in fixpoint form and abstraction

3. Determinism of multithreaded programs

- Formalization of a specific property

4. Domain and semantics of Java bytecode

- Low-level domain, specific alias analysis


5. Checkmate

- Generic sound analyzer of multithreaded programs

An example

Thread Deposit1

Variable	Value
tempt1	
...	




```
int tempt1=a.amount;  
tempt1=tempt1+1000$;  
a.amount=tempt1;
```

Shared Memory

Object	Field	Value
a	amount	10.000\$
	ID	58656
	...	
...		

Thread Deposit2

Variable	Value
tempt2	
...	



```
int tempt2=a.amount;  
tempt2=tempt2+1000$;  
a.amount=tempt2;
```

An example

Thread Deposit1

Variable	Value
tempt1	10.000\$
...	

```
int tempt1=a.amount;  
tempt1=tempt1+1000$;  
  
a.amount=tempt1;
```

Shared Memory

Object	Field	Value
a	amount	10.000\$
	ID	58656
	...	
...		

Thread Deposit2

Variable	Value
tempt2	
...	

```
int tempt2=a.amount;  
tempt2=tempt2+1000$;  
  
a.amount=tempt2;
```

An example

Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```

Shared Memory

Object	Field	Value
a	amount	10.000\$
	ID	58656
	...	
...		

Thread Deposit2

Variable	Value
tempt2	
...	

```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```

```
a.amount=tempt2;
```

An example

Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```

Shared Memory

Object	Field	Value
a	amount	11.000\$
	ID	58656
	...	
...		

Thread Deposit2

Variable	Value
tempt2	
...	

```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```

```
a.amount=tempt2;
```

An example


Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```



Shared Memory

Object	Field	Value
a	amount	11.000\$
	ID	58656
	...	
...		

Thread Deposit2

Variable	Value
tempt2	11.000\$
...	

```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```



```
a.amount=tempt2;
```

An example


Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```



Shared Memory

Object	Field	Value
a	amount	11.000\$
	ID	58656
	...	
...		


Thread Deposit2

Variable	Value
tempt2	12.000\$
...	

```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```

```
a.amount=tempt2;
```



An example


Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```



Shared Memory

Object	Field	Value
a	amount	12.000\$
	ID	58656
	...	
...		


Thread Deposit2

Variable	Value
tempt2	12.000\$
...	

```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```

```
a.amount=tempt2;
```



An example


Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```



Shared Memory

Object	Field	Value
a	amount	12.000\$
	ID	58656
	...	
...		


Thread Deposit2

Variable	Value
tempt2	12.000\$
...	

```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```

```
a.amount=tempt2;
```




At the end:

a.amount= 12.000\$

An example

Thread Deposit1

Variable	Value
tempt1	
...	




```
int tempt1=a.amount;  
tempt1=tempt1+1000$;  
a.amount=tempt1;
```

Shared Memory

Object	Field	Value
a	amount	10.000\$
	ID	58656
	...	
...		

Thread Deposit2

Variable	Value
tempt2	
...	



```
int tempt2=a.amount;  
tempt2=tempt2+1000$;  
a.amount=tempt2;
```

An example

Thread Deposit1

Variable	Value
tempt1	10.000\$
...	

```
int tempt1=a.amount;  
tempt1=tempt1+1000$;  
  
a.amount=tempt1;
```

Shared Memory

Object	Field	Value
a	amount	10.000\$
	ID	58656
	...	
...		

Thread Deposit2


Variable	Value
tempt2	
...	

```
int tempt2=a.amount;  
tempt2=tempt2+1000$;  
  
a.amount=tempt2;
```

An example

Thread Deposit1

Variable	Value
tempt1	10.000\$
...	

 `int tempt1=a.amount;`

`tempt1=tempt1+1000$;`

`a.amount=tempt1;`

Shared Memory

Object	Field	Value
a	amount	10.000\$
	ID	58656
	...	
...		

Thread Deposit2

Variable	Value
tempt2	10.000\$
...	

`int tempt2=a.amount;`

`tempt2=tempt2+1000$;`


`a.amount=tempt2;`

An example

Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;  
tempt1=tempt1+1000$;  
a.amount=tempt1;
```




Shared Memory

Object	Field	Value
a	amount	10.000\$
	ID	58656
	...	
...		

Thread Deposit2

Variable	Value
tempt2	10.000\$
...	

```
int tempt2=a.amount;  
tempt2=tempt2+1000$;  
a.amount=tempt2;
```



An example

Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```

Shared Memory

Object	Field	Value
a	amount	11.000\$
	ID	58656
	...	
...		

Thread Deposit2

Variable	Value
tempt2	10.000\$
...	

```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```

```
a.amount=tempt2;
```

An example


Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```



Shared Memory

Object	Field	Value
a	amount	11.000\$
	ID	58656
	...	
...		


Thread Deposit2

Variable	Value
tempt2	11.000\$
...	

```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```

```
a.amount=tempt2;
```



An example


Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```



Shared Memory

Object	Field	Value
a	amount	11.000\$
	ID	58656
	...	
...		


Thread Deposit2

Variable	Value
tempt2	11.000\$
...	

```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```

```
a.amount=tempt2;
```



An example


Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```



Shared Memory

Object	Field	Value
a	amount	11.000\$
	ID	58656
	...	
...		


Thread Deposit2

Variable	Value
tempt2	11.000\$
...	

```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```

```
a.amount=tempt2;
```



At the end:

a.amount=11.000\$

An example

Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```

Shared Memory

Object	Field	Value
a	amount	11.000\$
	ID	58656
	...	
...		

Thread Deposit2

Variable	Value
tempt2	11.000\$
...	

```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```

```
a.amount=tempt2;
```

1st execution: a.amount=12.000\$
2nd execution: a.amount=11.000\$

Our solution

- **Statically** analyze the determinism
 - > Focused on communications on shared memory
 - > **Generic** w.r.t.
 - Programming language
 - Numerical domain
 - Memory model
- **Advantages**
 - > Deal **directly** with the effects of arbitrary interleaving
 - > **Flexible**

Concrete domain and property

$$S : [\text{Var} \rightarrow (V \times \text{TId})]$$

- Each value is related to a thread identifier
 - > Trace **which thread wrote** it in the shared memory
- A program is not deterministic iff
 - > two executions
 - of the **same thread**
 - in the **same position** of the traces of execution
 - > contain two shared memories
 - in which the **same variable** contains values related to **different thread identifiers**

$$ds(s_1, s_2) = \text{false}$$



$$\exists \text{var} \in \text{dom}(s_1) \cap \text{dom}(s_2) : s_1(\text{var}) = (\text{val}_1, t_1),$$

$$s_2(\text{var}) = (\text{val}_2, t_2), t_1 \neq t_2$$

An example

Thread Deposit1:

Obj.	Field	Value	Thread
a	amount	10.000\$	System



Obj.	Field	Value	Thread
a	amount	11.000\$	Deposit1

Thread Deposit2:

Obj.	Field	Value	Thread
a	amount	11.000\$	Deposit1



Obj.	Field	Value	Thread
a	amount	12.000\$	Deposit2

Thread Deposit1:

Obj.	Field	Value	Thread
a	amount	10.000\$	System



Obj.	Field	Value	Thread
a	amount	11.000\$	Deposit1

Thread Deposit2:

Obj.	Field	Value	Thread
a	amount	10.000\$	System



Obj.	Field	Value	Thread
a	amount	11.000\$	Deposit2

An example

Thread Deposit1:

Obj.	Field	Value	Thread
a	amount	10.000\$	System



Obj.	Field	Value	Thread
a	amount	11.000\$	Deposit1

Thread Deposit2:

Obj.	Field	Value	Thread
a	amount	11.000\$	Deposit1



Obj.	Field	Value	Thread
a	amount	12.000\$	Deposit2

Thread Deposit1:

Obj.	Field	Value	Thread
a	amount	11.000\$	Deposit1

Non-deterministic executions!

Thread Deposit2:

Obj.	Field	Value	Thread
a	amount	10.000\$	System



Obj.	Field	Value	Thread
a	amount	11.000\$	Deposit2

Levels of abstraction

- First level: $\widehat{S} : [\text{Var} \rightarrow [\text{TId} \rightarrow \widehat{V}]]$
- Parameterized by an abstract numerical domain \widehat{V}
 - > One value for each thread
- Second level: $\overline{S} : [\text{Var} \rightarrow (\widehat{V} \times \wp(\text{TId}))]$
- Trace
 - > One abstract value
 - > The set of threads that may have written it
- Sound

$$\langle \wp(\Psi), \sqsubseteq \rangle \xrightleftharpoons[\alpha_\Psi]{\gamma_\Psi} \langle \widehat{\Psi}, \sqsubseteq_{\widehat{\Psi}} \rangle \xrightleftharpoons[\alpha_{\widehat{\Psi}}]{\gamma_{\widehat{\Psi}}} \langle \overline{\Psi}, \sqsubseteq_{\overline{\Psi}} \rangle$$

Determinism on abstract states

- First abstraction

$$\widehat{ds}(\widehat{\mathbf{s}}) = \text{false}$$



$$\exists \text{var} \in \text{dom}(\widehat{\mathbf{s}}) : |\text{dom}(\widehat{\mathbf{s}}(\text{var}))| > 1$$

- Second abstraction

$$\overline{ds}(\overline{\mathbf{s}}) = \text{false} \Leftrightarrow \exists \text{var} \in \text{dom}(\overline{\mathbf{s}}) : |\pi_2(\overline{\mathbf{s}}(\text{var}))| > 1$$

- Soundness

$$\forall \theta \in \wp(\Psi) : d(\theta) = \text{false} \Rightarrow \widehat{d}(\alpha_{\Psi}(\theta)) = \text{false}$$

$$\forall \mathbf{f} \in \widehat{\Psi} : \widehat{d}(\mathbf{f}) = \text{false} \Rightarrow \overline{d}(\alpha_{\widehat{\Psi}}(\mathbf{f})) = \text{false}$$

An example – 1st abstraction

Thread Deposit1:

Obj.	Field	Thread	Value
a	amount	System	[10.000..10.000]\$



Obj.	Field	Thread	Value
a	amount	Deposit1	[11.000..11.000]\$

Thread Deposit2:

Obj.	Field	Thread	Value
a	amount	System	[10.000..10.000]\$
		Deposit1	[11.000..11.000]\$



Obj.	Field	Thread	Value
a	amount	Deposit2	[11.000..12.000]\$

An example – 1st abstraction

Thread Deposit1:

Obj.	Field	Thread	Value
a	amount	System	[10.000..10.000]\$



Obj.	Field	Thread	Value
a	amount	Deposit1	[11.000..11.000]\$

Non-deterministic program!

Thread Deposit2:

Obj.	Field	Thread	Value
a	amount	System	[10.000..10.000]\$
		Deposit1	[11.000..11.000]\$



Obj.	Field	Thread	Value
a	amount	Deposit2	[11.000..12.000]\$

Weak determinism

$$\widehat{wds} : [\widehat{S} \rightarrow \{\text{true}, \text{false}\}]$$

$$\widehat{wds}(\widehat{s}) = \text{false}$$



$$\exists \text{var} \in \text{dom}(\widehat{s}) : |\text{dom}(\widehat{s}(\text{var}))| > 1$$

$$\wedge \exists t_1, t_2 \in \text{dom}(\widehat{s}(\text{var})) : \widehat{s}(\text{var})(t_1) \neq \widehat{s}(\text{var})(t_2)$$

- **Relax the full determinism**
 - > On the first level of abstraction
 - > Rely on a **numerical abstract domain**
- It allows non deterministic behaviors iff
 - > The abstract values written in parallel by different threads are the same
 - E.g. if the sign of the values is the same

An example – 1st abstraction

Thread Deposit1:

Obj.	Field	Thread	Value
a	amount	System	[10.000..10.000]\$



Obj.	Field	Thread	Value
a	amount	Deposit1	[11.000..11.000]\$

Thread Deposit2:

Obj.	Field	Thread	Value
a	amount	System	[10.000..10.000]\$
		Deposit1	[11.000..11.000]\$



Obj.	Field	Thread	Value
a	amount	Deposit2	[11.000..12.000]\$

An example – 1st abstraction

Thread Deposit1:

Obj.	Field	Thread	Value
a	amount	System	+



Obj.	Field	Thread	Value
a	amount	Deposit1	+

Thread Deposit2:

Obj.	Field	Thread	Value
a	amount	System	+
		Deposit1	+



Obj.	Field	Thread	Value
a	amount	Deposit2	+

Projecting states

- Check the determinism only
 - > On a subset of the shared variables, e.g.
 - Only the amount of the bank account

$$\alpha_S^{stPrj} : [S \rightarrow S]$$

$$\alpha_S^{stPrj}(s) = \{s' : \text{dom}(s') \subseteq \text{dom}(s), \forall \text{var} \in \text{dom}(s') : \\ s'(\text{var}) = s(\text{var}) \wedge \text{stPrj}(\text{var}) = \text{true}\}$$

$$dS_{stPrj} : [S \times S \rightarrow \{\text{true}, \text{false}\}]$$

$$dS_{stPrj}(s_1, s_2) = \text{false}$$



$$\exists \text{var} \in \text{dom}(\alpha_S^{stPrj}(s_1)) \cap \text{dom}(\alpha_S^{stPrj}(s_2)) : \\ s_1(\text{var}) = (\text{val}_1, t_1), s_2(\text{var}) = (\text{val}_2, t_2), t_1 \neq t_2$$

Projecting traces

- Check the determinism only
 - > On a subset of the trace
 - Only the actions that deposit or withdraw money

$$d_{trPrj} : [\wp(\Psi) \rightarrow \{\text{true}, \text{false}\}]$$

$$d_{trPrj}(\theta) = \text{false}$$



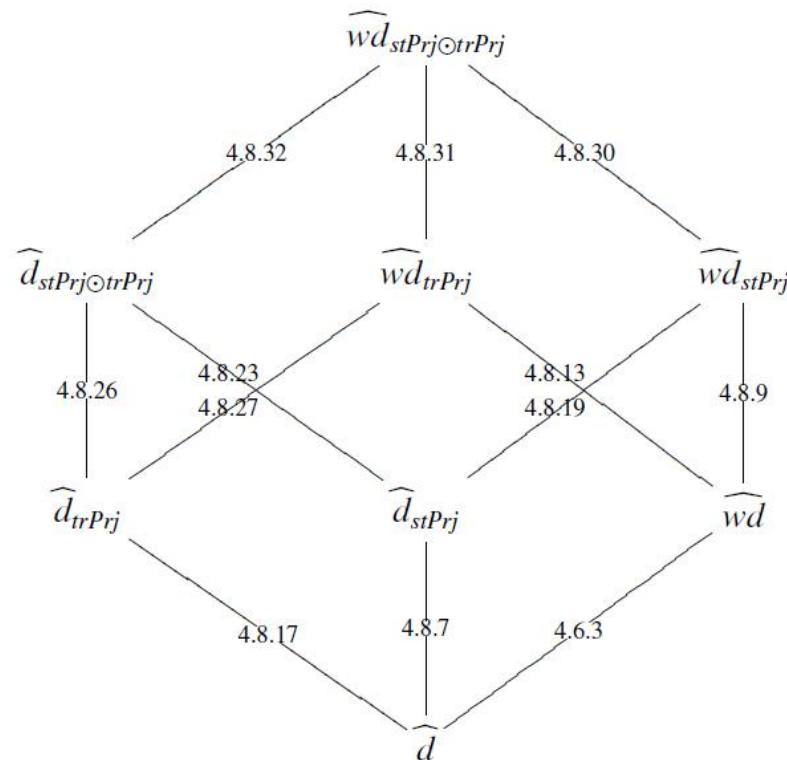
$$\exists f_1, f_2 \in \theta : \exists t \in \text{dom}(f_1) \cap \text{dom}(f_2) : \tau_1 = f_1(t), \tau_2 = f_2(t),$$

$$\exists i \in [0.. \min(\text{len}(\tau_1), \text{len}(\tau_2))] :$$

$$trPrj(i) = \text{true} \wedge ds(\tau_1(i), \tau_2(i)) = \text{false}$$

Global hierarchy

- Can be composed, e.g. check the determinism
 - > On a .amount
 - > When withdrawing
 - > If the values written are always positive



Semi-automatic parallelization

- Given a sequential program
 - > **Divide** it in two partitions - Input of the developer
 - > **Analyze** them as executed in parallel
 - > **Check** if the determinism is respected
 - Or one of its relaxations

Semi-automatic parallelization

- Given a sequential program
 - > **Divide** it in two partitions - Input of the developer
 - > **Analyze** them as executed in parallel
 - > **Check** if the determinism is respected
 - Or one of its relaxations

Thread 1 → `acc.deposit(1.000$);`
`acc.deposit(1.000$);` ← Thread 2

Non-determinism on `acc.amount`

Semi-automatic parallelization

- Given a sequential program
 - > **Divide** it in two partitions - Input of the developer
 - > **Analyze** them as executed in parallel
 - > **Check** if the determinism is respected
 - Or one of its relaxations

Thread 1 → `acc.deposit(1.000$);`
`acc.printAmount();` ← Thread 2

Non-determinism on stdout

Outline

1. Introduction

2. Happens-before memory model

- Definition in fixpoint form and abstraction

3. Determinism of multithreaded programs

- Formalization of a specific property

4. Domain and semantics of Java bytecode

- Low-level domain, specific alias analysis

5. Checkmate

- Generic sound analyzer of multithreaded programs

From theory to practice

- **Ultimate goal**
 - > Develop a static analysis of **Java** programs
- **Theoretical** approach:
 - > Set of thread identifiers
 - > Set of shared locations
 - > Set of synchronizable elements
- **From theory to... Java!**
 - > Threads: **objects**
 - > Shared memory: **heap**
 - > Synchronizable elements: monitors on **objects**

Features

- **We support**
 - > **Dynamic allocation** of shared memory
 - > **Dynamic creation and launch of threads**
 - > **Dynamic creation of monitors**
- **In addition, common Java features like**
 - > **Strings**
 - > **Arrays**
 - > **Static fields and methods**
 - > **Overload, overriding, recursion**
- **We fully support the Java bytecode language**

Concrete domain

- **Low-level domain**
 - > Simulate the Java Virtual Machine (JVM)
 - > Based on the JVM specification
 - Operand stack: $Op = \mathbb{ST}(\text{Val})$
 - Heap: $H : [\text{Ref} \rightarrow (\text{Obj} \cup \text{Arr} \cup \text{Str})]$
 - Local variables: $LV = \mathbb{AR}(\text{Val})$
 - Locked monitors: $L : [\text{Ref} \rightarrow \mathbb{N}]$
- We represent programs as **Control Flow Graph** CFG
- 1st abstraction:
 - > Executions on the Control Flow Graph exCFG
 - > **Sound** abstraction of real executions

$$\langle \wp(\Sigma^{\vec{r}}), \subseteq \rangle \xrightleftharpoons[\alpha_{\text{CFG}}]{\gamma_{\text{CFG}}} \langle \text{exCFG}, \sqsubseteq_{\text{CFG}} \rangle$$

Alias analysis

- Concrete references: potentially infinite
- We need to check when references
 - > May point to the same location (may-aliasing)
 - > Always point to the same location (must-aliasing)
- May aliasing:
 - > Bound each reference to the program point that allocates it
- Must aliasing:
 - > Each reference related to an equivalence class
 - > Rough but precise enough

Abstract domain and semantics

- Other components: (almost) pointwise abstraction
 - > Operand stack: $\overline{\text{Op}} = \text{ST}(\overline{\text{Val}})$
 - > Heap: $\overline{\text{H}} : [\overline{\text{P}} \rightarrow (\overline{\text{Obj}} \cup \overline{\text{Arr}} \cup \overline{\text{Str}})]$
 - > Local variables: $\overline{\text{LV}} = \text{AR}(\overline{\text{Val}})$
 - > Locked monitors: $\overline{\text{L}} : [\overline{\text{Ref}} \rightarrow \mathbb{N}]$

- Sound

$$\langle \wp(\Sigma), \sqsubseteq \rangle \xrightleftharpoons[\alpha_\Sigma]{\gamma_\Sigma} \langle \overline{\Sigma}, \sqsubseteq_\Sigma \rangle$$

- Operational semantics of statements
- Locally sound

$$\forall \sigma \in \Sigma : \alpha_\Sigma(\{\sigma' : \sigma \rightarrow \sigma'\}) \sqsubseteq_\Sigma \overline{\sigma'} : \alpha_\Sigma(\{\sigma\}) \overline{\rightarrow} \overline{\sigma'}$$

- Applied to HBMM and determinism

Outline

1. Introduction

2. Happens-before memory model

- Definition in fixpoint form and abstraction

3. Determinism of multithreaded programs

- Formalization of a specific property

4. Domain and semantics of Java bytecode

- Low-level domain, specific alias analysis

5. Checkmate

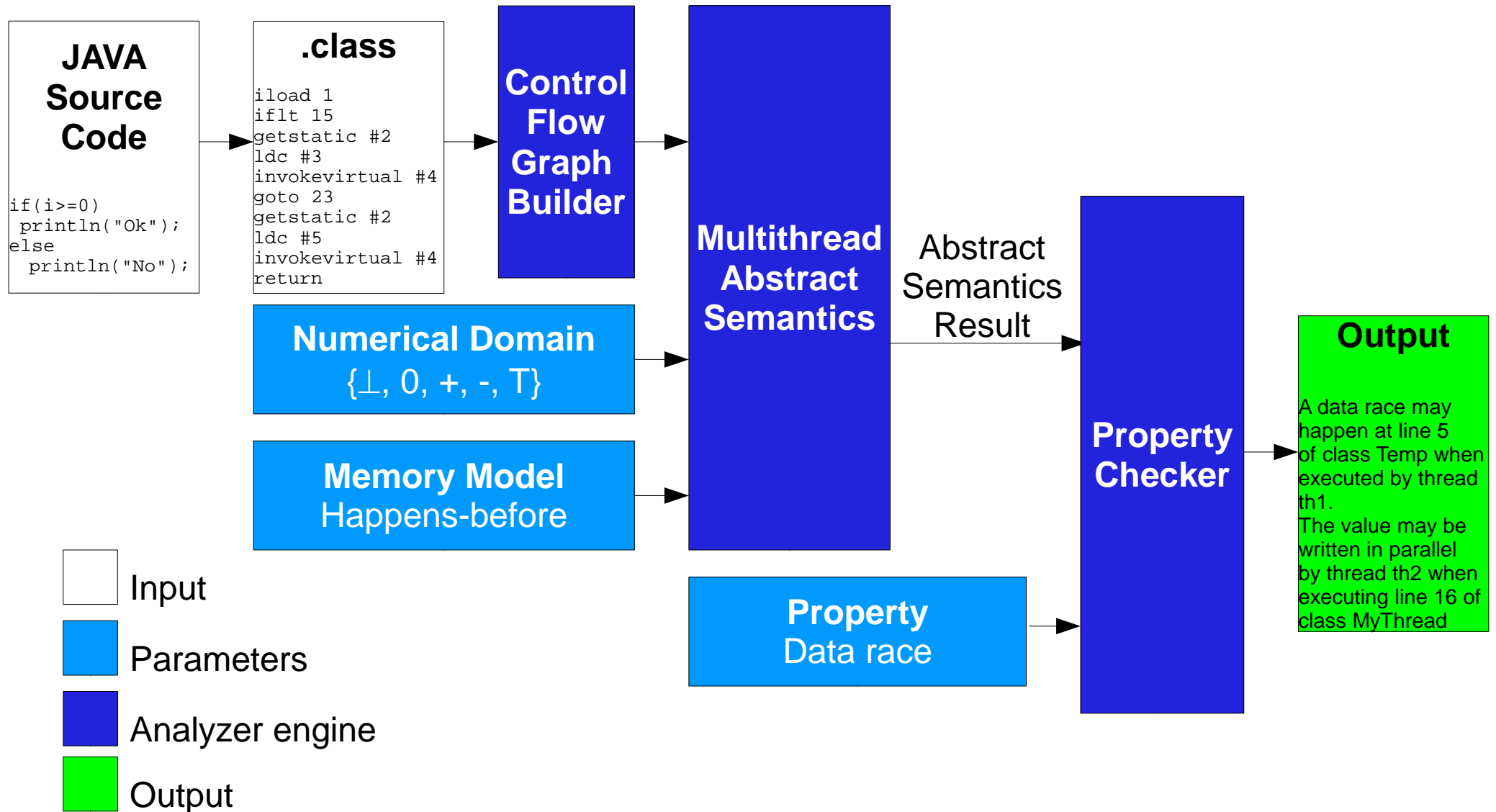
- Generic sound analyzer of multithreaded programs

Checkmate

- **Generic analyzer of multithreaded program**
 - > **Sound**
 - > **Flow-sensitive**
- **Generic w.r.t.**
 - > **Numerical domain**
 - Interval, sign, parity, congruence
 - > **Memory model**
 - Happens-before memory model
 - > **Property of interest**
 - Multithreading: data race, deadlock, determinism
 - Well-known: division by zero, access to null, etc..

<http://www.pietro.ferrara.name/checkmate>

Architecture



Experimental results

- Applied to
 - > A set of examples taken from [JMM]
 - > Some case studies taken from [LEA]
 - > A family of applications of increasing size
 - > Some benchmarks taken from [PRA, BENCH]
- Fast for small programs
- Precise
 - > But not scalable for large\ industrial programs

[JMM] J. Manson, W. Pugh, and S. V. Adve. *The Java memory model*. In ACM Press, editor, Proceedings of POPL '05, 2005.

[LEA] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1996.

[PRA] C. Von Praun and T. R. Gross. *Object race detection*. In ACM Press, editor, Proceedings of OOPSLA 01, 2001.

[BENCH] *Java Grande Forum Benchmark Suite*. At <http://www.epcc.ed.ac.uk/research/activities/java-grande/>

Incremental application

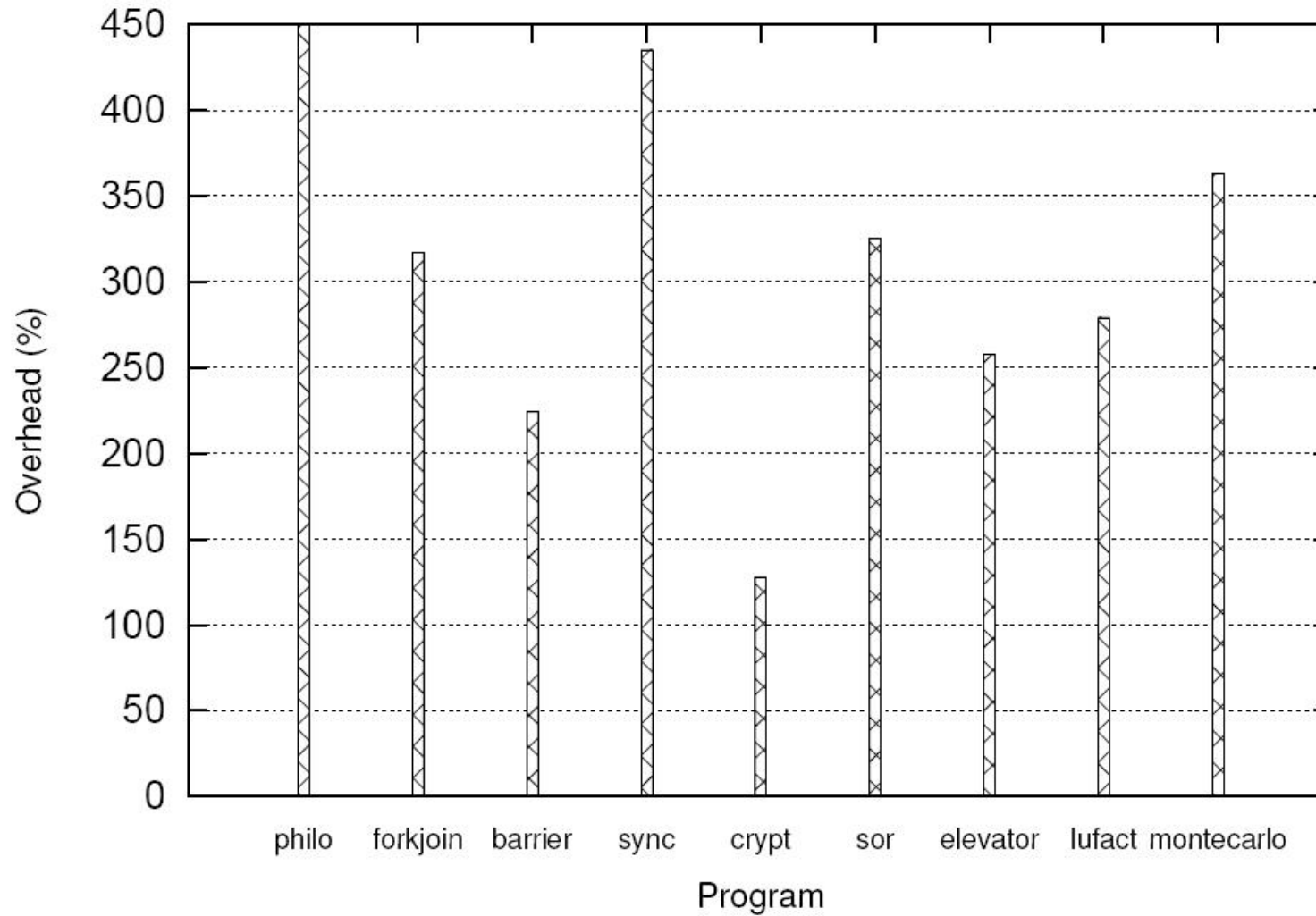
	Th.	St.	Top	Sign	Int.	Par.	Cong.
1	3	452	814	361	217	404	294
2	5	684	409	391	356	620	545
3	7	807	712	595	925	521	642
4	9	1049	799	823	3806	703	642
5	11	1173	1090	919	5887	779	616
6	13	1405	1382	824	7161	900	986
7	15	1526	1071	1647	9289	1340	863
8	17	1758	1018	1269	10999	1263	1221
9	20	1878	1421	2212	11691	1274	1623
10	24	2294	1466	2432	17016	863	1906

External benchmarks

Program	St.	Th.	Top	Sign	Int.	Par.	Cong.
philo	213	2	<1''	<1''	1''	<1''	<1''
forkjoin	170	2	<1''	<1''	<1''	<1''	<1''
barrier	363	3	<1''	1''	2''	1''	1''
sync	320	3	1''	1''	3''	1''	2''
crypt	2636	3	5''	6''	17''	6''	5''
sor	1121	2	4''	7''	17''	6''	5''
elevator	1829	2	31''	11''	19''	30''	29''
lufact	3732	2	27''	53''	5'59''	29''	29''
montecarlo	3864	2	1'02''	2'35''	1h00'56''	1'43''	1'04''

Overhead of multithread semantics

Overhead in % of multithread fixpoint computation using Intervals and HB memory model



Conclusion

- **Generic static analysis of multithreaded programs**
 - > Based on **abstract interpretation**
 - > Applied to a **real** programming language
 - **Bytecode** level, it can analyze other languages
 - > E.g. Scala
 - > **Implementation**
 - Experimental results encouraging
 - **Scalability** is still an open **issue**
- **Other generic analyzers scale up**
 - > **Local reasoning**, i.e. not whole program analyses
 - > Based on method boundary annotations

Future work

- How to **refine** the **MM** and its analysis
 - > Other synchronizations have to be considered
 - > Interesting restrictions of the Java MM
- How to **relax** the property of determinism
- **Refine** bytecode domain and semantics
 - > Goal: apply to numerical **relational domains**
- **Implement** it in Checkmate

Future work

- Whole program analysis
 - > Limit: it does **not scale!**
- **Modular reasoning**
 - > **Impossible** on multithreaded programs
 - > **Lack** of programming languages and contracts
- **Object-oriented programs**
 - > Restrict the **visibility** of fields and methods
 - public, private, protected
 - > **Contracts** on classes and methods
- **Intuition**
 - > **Apply and tune these ideas to multithreading**

PhD Thesis

[Ferr09] P. Ferrara, "*Static analysis via abstract interpretation of multithreaded programs*", PhD Thesis, Ecole Polytechnique of Paris (France) and University "Ca' Foscari" of Venice (Italy), May 2009

- The work I presented is my **PhD thesis**
 - > Supervisors:
 - **Radhia Cousot** (CNRS/ENS)
 - **Agostino Cortesi** (University "Ca' Foscari" of Venice)
 - > There is **something more** in it
 - Application of a generic static analyzer (Clousot @ Microsoft) to the analysis of buffer overrun
 - OOPSLA '08
- It will be soon (in the next **few** days) on my website <http://www.pietro.ferrara.name>

Pietro Ferrara: "Static analysis via abstract interpretation of multithreaded programs"

IRISA-INRIA, Rennes, France

Publications

[Ferr08] P. Ferrara, "*Static analysis via abstract interpretation of the happens-before memory model*", in Springer editor, Proceedings of TAP 2008, volume 4966 of LNCS, Prato, Italy, April 9-11, 2008

[Ferr08b] P. Ferrara, "*Static analysis of the determinism of multithreaded programs*", in IEEE Computer Society, editor, Proceedings of SEFM 2008, Cape Town, South Africa, November 10-14, 2008

[Ferr08a] P. Ferrara, "*A fast and precise analysis for data race detection*", in Elsevier editor, Proceedings of Bytecode'08, ENTCS, Budapest, Hungary, April 6, 2008

[Ferr09a] P. Ferrara, "*Checkmate: a Generic Static Analyzer of Java Multithreaded Programs*", in IEEE Computer Society, editor, Proceedings of SEFM 2009, Hanoi, Vietnam, November, 2009

Pietro Ferrara: "Static analysis via abstract interpretation of multithreaded programs"

IRISA-INRIA, Rennes, France

Question time

Thank you!