

# Static Analysis via Abstract Interpretation of the Happens-Before Memory Model

Pietro Ferrara

Ecole Polytechnique, Paris  
F-91128 Palaiseau (France)  
Università Ca' Foscari di Venezia  
I-30170 Venezia (Italy)  
[Pietro.Ferrara@polytechnique.edu](mailto:Pietro.Ferrara@polytechnique.edu)  
[ferrara@dsi.unive.it](mailto:ferrara@dsi.unive.it)

May 28, 2008

- ▶ This talk presents mostly the work published by:  
**P. Ferrara** "*Static analysis via abstract interpretation of the happens-before memory model*", in Springer, editor, Proceedings of the Second International Conference on Tests and Proofs (TAP 2008), volume 4966 of *Lecture Notes in Computer Science*, Prato, Italy, April 9-11, 2008
- ▶ In addition: implementation and experimental results
- ▶ The implementation is based on the work presented by:  
[FER08] **Pietro Ferrara** "*A fast and precise analysis for data race detection*", in Elsevier, editor, Proceedings of Bytecode 08, volume *ENTCS*, Budapest, Hungary, April 5, 2008

# Outline

Memory Model

Concrete domain and semantics

Abstraction

Demo

Experimental results

Conclusion

- ▶ Multicore architectures
  - ▶ many parallel tasks
- ▶ More difficult than sequential programs [SUT05]
  - ▶ interleaving of the executions of different threads
  - ▶ implicit communications through the shared memory
  - ▶ specification of the memory model [LEE06]

# Sequential programs

- ▶ Execution represented as a trace of states:  
1 : `for(int i = 0; i < arr.length; i ++)`  
2 : `arr[i] = 0;`
- ▶ Represented by  $\sigma_0 \xrightarrow{1} \sigma_1 \xrightarrow{2} \sigma_2 \xrightarrow{1} \dots$

# Sequential programs

- ▶ Execution represented as a trace of states:  
1 : `for(int i = 0; i < arr.length; i ++)`  
2 : `arr[i] = 0;`
- ▶ Represented by  $\sigma_0 \xrightarrow{1} \sigma_1 \xrightarrow{2} \sigma_2 \xrightarrow{1} \dots$
- ▶ We know exactly the sequence of statements executed and the order in which they are executed

# Multithreaded programs

System	MyThread
<pre>S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i &lt; a.length; i ++) S4 : a[i] = 0;</pre>	<pre>public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j &lt; a.length; j ++) T3 : sum += a[j]; }</pre>

- ▶ If executed on a single core architecture:

# Multithreaded programs

System	MyThread
S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i < a.length; i ++) S4 : a[i] = 0;	public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j < a.length; j ++) T3 : sum += a[j]; }

- ▶ If executed on a single core architecture:

$\sigma_0$   $\xrightarrow{S1: System}$   $\sigma_1$

# Multithreaded programs

System	MyThread
S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i < a.length; i ++) S4 : a[i] = 0;	public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j < a.length; j ++) T3 : sum+ = a[j]; }

- ▶ If executed on a single core architecture:

$\sigma_0 \xrightarrow{S1: System} \sigma_1 \xrightarrow{S2: System} \sigma_2$

# Multithreaded programs

System	MyThread
<pre>S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i &lt; a.length; i++) S4 : a[i] = 0;</pre>	<pre>public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j &lt; a.length; j++) T3 : sum += a[j]; }</pre>

- ▶ If executed on a single core architecture:

$\sigma_0$      $\xrightarrow{S1: System}$      $\sigma_1$      $\xrightarrow{S2: System}$      $\sigma_2$      $\xrightarrow{T1: MyThread}$      $\sigma_3$

# Multithreaded programs

System	MyThread
S1 : <code>t = new MyThread(a);</code> S2 : <code>t.start();</code> S3 : <code>for(int i = 0; i &lt; a.length; i ++)</code> S4 : <code>  a[i] = 0;</code>	<code>public void run(){</code> T1 : <code>  int sum = 0;</code> T2 : <code>  for(int j = 0; j &lt; a.length; j ++)</code> T3 : <code>    sum += a[j];</code> }

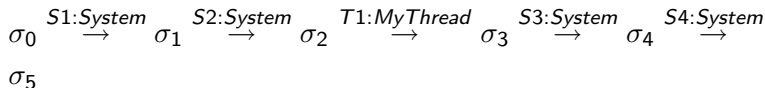
- ▶ If executed on a single core architecture:

$\sigma_0 \xrightarrow{S1: System} \sigma_1 \xrightarrow{S2: System} \sigma_2 \xrightarrow{T1: MyThread} \sigma_3 \xrightarrow{S3: System} \sigma_4$

# Multithreaded programs

System	MyThread
S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i < a.length; i ++) S4 : a[i] = 0;	public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j < a.length; j ++) T3 : sum += a[j]; }

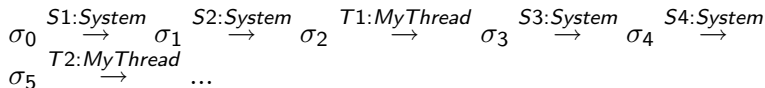
- ▶ If executed on a single core architecture:



# Multithreaded programs

System	MyThread
S1 : t = new MyThread(a);	public void run(){
S2 : t.start();	T1 : int sum = 0;
S3 : for(int i = 0; i < a.length; i ++)	T2 : for(int j = 0; j < a.length; j ++)
S4 : a[i] = 0;	T3 : sum += a[j];
	}

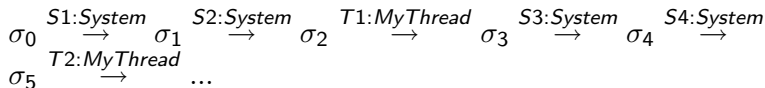
- ▶ If executed on a single core architecture:



# Multithreaded programs

System	MyThread
S1 : t = new MyThread(a);	public void run(){
S2 : t.start();	T1 : int sum = 0;
S3 : for(int i = 0; i < a.length; i ++)	T2 : for(int j = 0; j < a.length; j ++)
S4 : a[i] = 0;	T3 : sum += a[j];
	}

- ▶ If executed on a single core architecture:



- ▶ Casual interleaving between the executions of different threads

# Multithreaded programs

System	MyThread
<pre>S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i &lt; a.length; i ++) S4 : a[i] = 0;</pre>	<pre>public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j &lt; a.length; j ++) T3 : sum+ = a[j]; }</pre>

► Dual-core architecture:

Core 1:  $\sigma_0$

Core 2:  $\sigma_0$

# Multithreaded programs

System	MyThread
<pre>S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i &lt; a.length; i ++ ) S4 : a[i] = 0;</pre>	<pre>public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j &lt; a.length; j ++ ) T3 : sum+ = a[j]; }</pre>

► Dual-core architecture:

Core 1:  $\sigma_0 \xrightarrow{S1: System} \sigma_1$

Core 2:  $\sigma_0$

# Multithreaded programs

System	MyThread
<pre>S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i &lt; a.length; i ++ ) S4 : a[i] = 0;</pre>	<pre>public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j &lt; a.length; j ++ ) T3 : sum+ = a[j]; }</pre>

► Dual-core architecture:

Core 1:  $\sigma_0 \xrightarrow{S1: \text{System}} \sigma_1 \xrightarrow{S2: \text{System}} \sigma_2$

Core 2:  $\sigma_0$

# Multithreaded programs

System	MyThread
<pre>S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i &lt; a.length; i++) S4 : a[i] = 0;</pre>	<pre>public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j &lt; a.length; j++) T3 : sum += a[j]; }</pre>

► Dual-core architecture:

Core 1:  $\sigma_0 \xrightarrow{S1: \text{System}} \sigma_1 \xrightarrow{S2: \text{System}} \sigma_2$

Core 2:  $\sigma_0 \xrightarrow{T1: \text{MyThread}} \sigma'_1$

# Multithreaded programs

System	MyThread
S1 : <code>t = new MyThread(a);</code> S2 : <code>t.start();</code> S3 : <code>for(int i = 0; i &lt; a.length; i ++)</code> S4 : <code>  a[i] = 0;</code>	<code>public void run(){</code> T1 : <code>int sum = 0;</code> T2 : <code>for(int j = 0; j &lt; a.length; j ++)</code> T3 : <code>  sum+ = a[j];</code> <code>}</code>

► Dual-core architecture:

Core 1:  $\sigma_0 \xrightarrow{S1: \text{System}} \sigma_1 \xrightarrow{S2: \text{System}} \sigma_2 \xrightarrow{S3: \text{System}} \sigma_3$

Core 2:  $\sigma_0 \xrightarrow{T1: \text{MyThread}} \sigma'_1 \xrightarrow{T2: \text{MyThread}} \sigma'_2$

# Multithreaded programs

System	MyThread
<pre>S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i &lt; a.length; i ++ ) S4 : a[i] = 0;</pre>	<pre>public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j &lt; a.length; j ++ ) T3 : sum+ = a[j]; }</pre>

► Dual-core architecture:

Core 1:  $\sigma_0 \xrightarrow{S1: \text{System}} \sigma_1 \xrightarrow{S2: \text{System}} \sigma_2 \xrightarrow{S3: \text{System}} \sigma_3 \xrightarrow{S4: \text{System}} \dots$

Core 2:  $\sigma_0 \xrightarrow{T1: \text{MyThread}} \sigma'_1 \xrightarrow{T2: \text{MyThread}} \sigma'_2$

# Multithreaded programs

System	MyThread
<pre>S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i &lt; a.length; i ++ ) S4 : a[i] = 0;</pre>	<pre>public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j &lt; a.length; j ++ ) T3 : sum+ = a[j]; }</pre>

► Dual-core architecture:

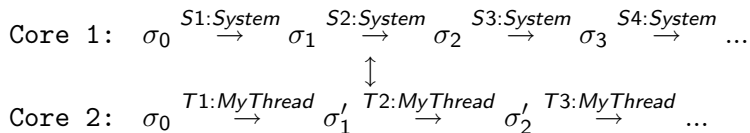
Core 1:  $\sigma_0 \xrightarrow{S1: \text{System}} \sigma_1 \xrightarrow{S2: \text{System}} \sigma_2 \xrightarrow{S3: \text{System}} \sigma_3 \xrightarrow{S4: \text{System}} \dots$

Core 2:  $\sigma_0 \xrightarrow{T1: \text{MyThread}} \sigma'_1 \xrightarrow{T2: \text{MyThread}} \sigma'_2 \xrightarrow{T3: \text{MyThread}} \dots$

# Multithreaded programs

System	MyThread
S1 : <code>t = new MyThread(a);</code> S2 : <code>t.start();</code> S3 : <code>for(int i = 0; i &lt; a.length; i ++)</code> S4 : <code>  a[i] = 0;</code>	<code>public void run(){</code> T1 : <code>  int sum = 0;</code> T2 : <code>  for(int j = 0; j &lt; a.length; j ++)</code> T3 : <code>    sum+ = a[j];</code> }

► Dual-core architecture:



# Multithreaded programs

System	MyThread
<pre>S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i &lt; a.length; i ++) S4 :   a[i] = 0;</pre>	<pre>public void run(){   T1 : int sum = 0;   T2 : for(int j = 0; j &lt; a.length; j ++)   T3 :   sum+ = a[j]; }</pre>

► Dual-core architecture:

Core 1:  $\sigma_0$

Core 2:  $\sigma_0$

# Multithreaded programs

System	MyThread
<pre>S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i &lt; a.length; i ++) S4 : a[i] = 0;</pre>	<pre>public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j &lt; a.length; j ++) T3 : sum+ = a[j]; }</pre>

► Dual-core architecture:

Core 1:  $\sigma_0$

Core 2:  $\sigma_0 \xrightarrow{S1: System} \sigma'_1$

# Multithreaded programs

System	MyThread
<pre>S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i &lt; a.length; i ++ ) S4 : a[i] = 0;</pre>	<pre>public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j &lt; a.length; j ++ ) T3 : sum+ = a[j]; }</pre>

► Dual-core architecture:

Core 1:  $\sigma_0 \xrightarrow{S2: System} \sigma_2$

Core 2:  $\sigma_0 \xrightarrow{S1: System} \sigma'_1$

# Multithreaded programs

System	MyThread
<pre>S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i &lt; a.length; i++) S4 : a[i] = 0;</pre>	<pre>public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j &lt; a.length; j++) T3 : sum += a[j]; }</pre>

► Dual-core architecture:

Core 1:  $\sigma_0 \xrightarrow{S2: System} \sigma_2 \xrightarrow{T1: MyThread} \sigma_3$

Core 2:  $\sigma_0 \xrightarrow{S1: System} \sigma'_1$

# Multithreaded programs

System	MyThread
S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i < a.length; i ++) S4 : a[i] = 0;	public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j < a.length; j ++) T3 : sum+ = a[j]; }

► Dual-core architecture:

Core 1:  $\sigma_0 \xrightarrow{S2: System} \sigma_2 \xrightarrow{T1: MyThread} \sigma_3 \xrightarrow{T2: MyThread} \sigma_4$

Core 2:  $\sigma_0 \xrightarrow{S1: System} \sigma'_1 \xrightarrow{S3: System} \sigma'_2$

# Multithreaded programs

System	MyThread
<pre>S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i &lt; a.length; i++) S4 : a[i] = 0;</pre>	<pre>public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j &lt; a.length; j++) T3 : sum += a[j]; }</pre>

► Dual-core architecture:

Core 1:  $\sigma_0 \xrightarrow{S2: System} \sigma_2 \xrightarrow{T1: MyThread} \sigma_3 \xrightarrow{T2: MyThread} \sigma_4$

Core 2:  $\sigma_0 \xrightarrow{S1: System} \sigma'_1 \xrightarrow{S3: System} \sigma'_2 \xrightarrow{S4: System} \dots$

# Multithreaded programs

System	MyThread
<pre>S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i &lt; a.length; i++) S4 : a[i] = 0;</pre>	<pre>public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j &lt; a.length; j++) T3 : sum += a[j]; }</pre>

## ► Dual-core architecture:

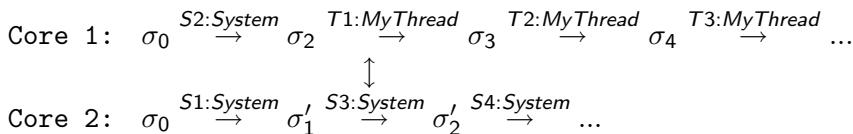
Core 1:  $\sigma_0 \xrightarrow{S2: \text{System}} \sigma_2 \xrightarrow{T1: \text{MyThread}} \sigma_3 \xrightarrow{T2: \text{MyThread}} \sigma_4 \xrightarrow{T3: \text{MyThread}} \dots$

Core 2:  $\sigma_0 \xrightarrow{S1: \text{System}} \sigma'_1 \xrightarrow{S3: \text{System}} \sigma'_2 \xrightarrow{S4: \text{System}} \dots$

# Multithreaded programs

System	MyThread
S1 : <code>t = new MyThread(a);</code> S2 : <code>t.start();</code> S3 : <code>for(int i = 0; i &lt; a.length; i ++)</code> S4 : <code>a[i] = 0;</code>	<code>public void run(){</code> T1 : <code>int sum = 0;</code> T2 : <code>for(int j = 0; j &lt; a.length; j ++)</code> T3 : <code>sum+ = a[j];</code> <code>}</code>

## ► Dual-core architecture:



# Memory model

- ▶ Many architectures, many compilers, ...
- ▶ What is legal and what is not?

# Memory model

- ▶ Many architectures, many compilers, ...
- ▶ What is legal and what is not?
- ▶ The memory model defines which multithreaded executions are legal and which are not!

# Memory model

- ▶ Many architectures, many compilers, ...
- ▶ What is legal and what is not?
- ▶ The memory model defines which multithreaded executions are legal and which are not!
  - ▶ Memory Model  $\supseteq$  executions on single core architectures
  - ▶ Memory Model  $\supseteq$  executions on dual core architectures
  - ▶ ...

# Memory model

- ▶ Many architectures, many compilers, ...
- ▶ What is legal and what is not?
- ▶ The memory model defines which multithreaded executions are legal and which are not!
  - ▶ Memory Model  $\supseteq$  executions on single core architectures
  - ▶ Memory Model  $\supseteq$  executions on dual core architectures
  - ▶ ...
- ▶ It does not define HOW a multithreaded program is executed
- ▶ It defines WHAT a multithreaded program is allowed to see
  - ▶ i.e.: which values can be read from the shared memory

# Memory Model

- ▶ Two opposite needs:
  - ▶ restrict the non-deterministic behaviors
  - ▶ allow as many compiler optimizations as possible
- ▶ e.g. sequential consistency:
  - ▶ “...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program” [LAM79]
- ▶ too much restrictive in practice, it does not allow the most part of optimizations

## An example

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

- ▶ Supposing that at the beginning  $i = 0, j = 0$

## An example

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

- ▶ Supposing that at the beginning  $i = 0, j = 0$
- ▶ May the exception be thrown?

# An example

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

- ▶ Supposing that at the beginning  $i = 0, j = 0$
- ▶ May the exception be thrown?
  - ▶ sequential consistency: NO!

# An example

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

- ▶ Supposing that at the beginning  $i = 0, j = 0$
- ▶ May the exception be thrown?
  - ▶ sequential consistency: NO!
  - ▶ practical needs: YES!

# An example

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

- ▶ Supposing that at the beginning  $i = 0, j = 0$
- ▶ May the exception be thrown?
  - ▶ sequential consistency: NO!
  - ▶ practical needs: YES!
    - ▶ swap of independent statements

# The Happens-Before Memory Model

- ▶ Program order:
  - ▶ intra-thread order of statements
- ▶ Synchronizes-with relation: involve two **synchronized** actions.  
E.g. the acquisition and the release of a monitor
- ▶ Happens-before order [LAM78]:  $a_1$  happens-before  $a_2$  if
  - ▶  $a_1$  appears before  $a_2$  in the program order
  - ▶  $a_2$  synchronizes-with  $a_1$
  - ▶ you can reach  $a_2$  by following happens-before edges starting from  $a_1$

- ▶ Consistency rule:
  - ▶ a read  $r$  of a variable  $v$  is allowed to see a write  $w$  to  $v$  if:
    - ▶  $r$  does not happens-before  $w$
    - ▶ there is not any  $w'$  to  $v$  that happens-before  $r$  and such that  $w$  happens-before it

- ▶ Consistency rule:
  - ▶ a read  $r$  of a variable  $v$  is allowed to see a write  $w$  to  $v$  if:
    - ▶  $r$  does not happens-before  $w$
    - ▶ there is not any  $w'$  to  $v$  that happens-before  $r$  and such that  $w$  happens-before it
- ▶ The happens-before memory model is an approximation of the Java one
  - ▶ If a behavior is allowed by the Java MM, it is allowed also by the happens-before MM

# Synchronizes-with

- ▶ Many different ways of synchronization:
  - ▶ mutual exclusion on monitors
  - ▶ wait and notify on objects
  - ▶ the first action of a thread synchronizes-with the statement that launched it
  - ▶ ...

# Synchronizes-with

- ▶ Many different ways of synchronization:
  - ▶ mutual exclusion on monitors
  - ▶ wait and notify on objects
  - ▶ the first action of a thread synchronizes-with the statement that launched it
  - ▶ ...
- ▶ We are generic w.r.t. the programming language
- ▶ We focus on
  - ▶ mutual exclusion
  - ▶ launch of a thread

# Synchronizes-with

- ▶ Many different ways of synchronization:
  - ▶ mutual exclusion on monitors
  - ▶ wait and notify on objects
  - ▶ the first action of a thread synchronizes-with the statement that launched it
  - ▶ ...
- ▶ We are generic w.r.t. the programming language
- ▶ We focus on
  - ▶ mutual exclusion
  - ▶ launch of a thread
- ▶ Other synchronizations... future work!

# Our contribution

- ▶ A **static analysis**
- ▶ On all the multithreaded programs
- ▶ Sound w.r.t. the happens-before memory model
- ▶ Based on the **abstract interpretation** theory

# Outline

Memory Model

Concrete domain and semantics

Abstraction

Demo

Experimental results

Conclusion

# Single-thread Domain

- ▶ Generic w.r.t. the programming language ...

# Single-thread Domain

- ▶ Generic w.r.t. the programming language ...
- ▶ ... we do not formally define in the details the concrete domain!

# Single-thread Domain

- ▶ Generic w.r.t. the programming language ...
- ▶ ... we do not formally define in the details the concrete domain!
- ▶ Some functions provide us the essential information
  - ▶  $shared : St \mapsto Sh$
  - ▶  $action : St \mapsto \perp_a \cup (\{r, w\} \times Loc \times Val)$
  - ▶  $synchronised : St \mapsto \wp(Sync)$
- ▶ Parameterized also on the intra-thread transfer function  $\xrightarrow{\circ}$   
[FER08] defines a concrete domain and the semantics of the intra-thread transfer function on a subset of the Java bytecode language

- ▶ For each thread we collect the trace representing its execution

$$\Psi : TId \rightarrow St^{\vec{t}}$$

- ▶ For each thread we collect the trace representing its execution

$$\Psi : TId \rightarrow St^{\vec{t}}$$

$$\Omega : TId \rightarrow ((TId \times Integer) \cup \perp_{\Omega})$$

- ▶  $t_1 \mapsto (t_2, 5)$  means that thread  $t_1$  has been launched by thread  $t_2$  at the instruction at program counter 5

- ▶ For each thread we collect the trace representing its execution

$$\Psi : TId \rightarrow St^{\vec{t}}$$

$$\Omega : TId \rightarrow ((TId \times Integer) \cup \perp_{\Omega})$$

- ▶  $t_1 \mapsto (t_2, 5)$  means that thread  $t_1$  has been launched by thread  $t_2$  at the instruction at program counter 5
- ▶ We abstract away the inter-thread order of execution!

# Multithreaded programs

System	MyThread
S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i < a.length; i ++) S4 : a[i] = 0;	public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j < a.length; j ++) T3 : sum += a[j]; }

►  $\Psi$  :

$$\text{System} \mapsto \{\sigma_0 \xrightarrow{S1} \sigma_1 \xrightarrow{S2} \sigma_2 \xrightarrow{S3} \sigma_3 \xrightarrow{S4} \sigma_4 \xrightarrow{S3} \sigma_5 \xrightarrow{S4} \dots\}$$
$$\text{MyThread} \mapsto \{\sigma'_0 \xrightarrow{T1} \sigma'_1 \xrightarrow{T2} \sigma'_2 \xrightarrow{T3} \sigma'_3 \xrightarrow{T2} \sigma'_4 \xrightarrow{T3} \sigma'_5 \xrightarrow{T2} \dots\}$$

# Multithreaded programs

System	MyThread
S1 : t = new MyThread(a); S2 : t.start(); S3 : for(int i = 0; i < a.length; i ++) S4 : a[i] = 0;	public void run(){ T1 : int sum = 0; T2 : for(int j = 0; j < a.length; j ++) T3 : sum += a[j]; }

►  $\Psi$  :

$$\begin{aligned}\text{System} &\mapsto \{\sigma_0 \xrightarrow{S1} \sigma_1 \xrightarrow{S2} \sigma_2 \xrightarrow{S3} \sigma_3 \xrightarrow{S4} \sigma_4 \xrightarrow{S3} \sigma_5 \xrightarrow{S4} \dots\} \\ \text{MyThread} &\mapsto \{\sigma'_0 \xrightarrow{T1} \sigma'_1 \xrightarrow{T2} \sigma'_2 \xrightarrow{T3} \sigma'_3 \xrightarrow{T2} \sigma'_4 \xrightarrow{T3} \sigma'_5 \xrightarrow{T2} \dots\}\end{aligned}$$

►  $\Omega : \{\text{MyThread} \mapsto (\text{System}, S2), \text{System} \mapsto \perp_{\Omega}\}$

# Intra-thread semantics

- ▶ *visible* function: return all the values written in parallel by other threads that may be seen following the HB memory model
- ▶ *step* function: defines a single step in the computation. Read from the shared memory one of the visible values  $\rightarrow$  nondeterministic choices

## Definition ( $\mathbb{S}^\circ$ )

$$\mathbb{S}^\circ : \Psi \times \Omega \times TId \mapsto \wp(St^{\vec{T}})$$

$$\mathbb{S}^\circ \llbracket f, r, t \rrbracket = \text{lfp}_{\emptyset}^{\subseteq} \lambda T. \{\sigma_0\} \cup \{\sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \rightarrow \sigma_i : \\ \sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \in T \wedge \sigma_i \in \text{step}(t, f, r)\}$$

# Multithread semantics

- ▶ Each time we compute the single thread semantics of a thread it may expose new values to other threads
- ▶ Iterates the single-thread semantics until a fixpoint is reached

## Definition ( $S^{\parallel}$ )

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

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

# An example

System	Thread 1	Thread 2
<code>new Thread1().start();</code> <code>new Thread2().start();</code>	<code>i = 1;</code> <code>j = 1;</code>	<code>if(j == 1&amp;&amp; i == 0)</code> <code>throw new Exception();</code>

# An example - 1st iteration

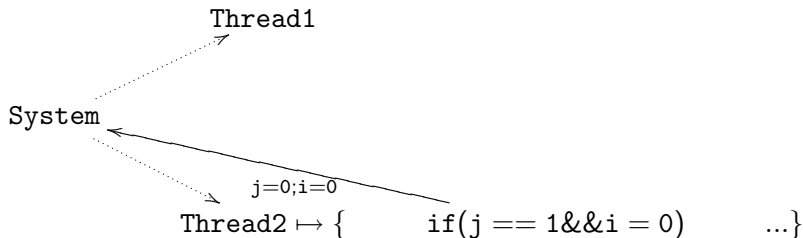
System	Thread 1	Thread 2
<code>new Thread1().start();</code>	<code>i = 1;</code>	<code>if(j == 1&amp;&amp; i == 0)</code>
<code>new Thread2().start();</code>	<code>j = 1;</code>	<code>throw new Exception();</code>

## System

- ▶ The previous state is empty
- ▶ Only an active thread: the one launched by the system!
- ▶ We compute only the semantics of that thread

## An example - 2nd iteration

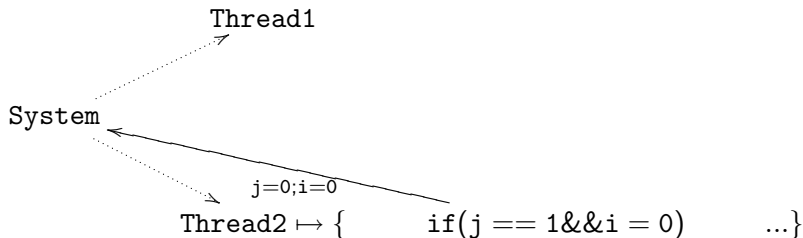
System	Thread 1	Thread 2
<code>new Thread1().start();</code>	<code>i = 1;</code>	<code>if(j == 1&amp;&amp; i == 0)</code>
<code>new Thread2().start();</code>	<code>j = 1;</code>	<code>throw new Exception();</code>



- ▶ We compute the semantics of three threads
- ▶ ... but the previous state exposes only the values written by System!

## An example - 2nd iteration

System	Thread 1	Thread 2
<code>new Thread1().start();</code>	<code>i = 1;</code>	<code>if(j == 1&amp;&amp; i == 0)</code>
<code>new Thread2().start();</code>	<code>j = 1;</code>	<code>throw new Exception();</code>

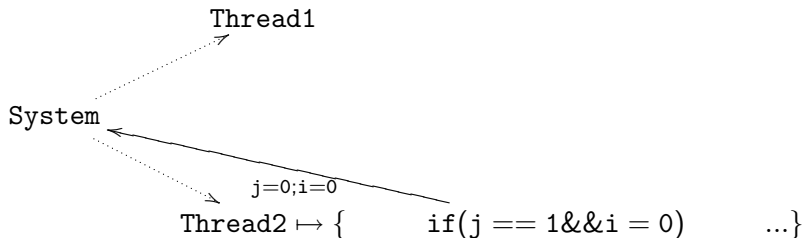


- ▶ We compute the semantics of three threads
- ▶ ... but the previous state exposes only the values written by System!

`visible(j) = {0}`

## An example - 2nd iteration

System	Thread 1	Thread 2
<code>new Thread1().start();</code> <code>new Thread2().start();</code>	<code>i = 1;</code> <code>j = 1;</code>	<code>if(j == 1 &amp;&amp; i == 0)</code> <code>throw new Exception();</code>



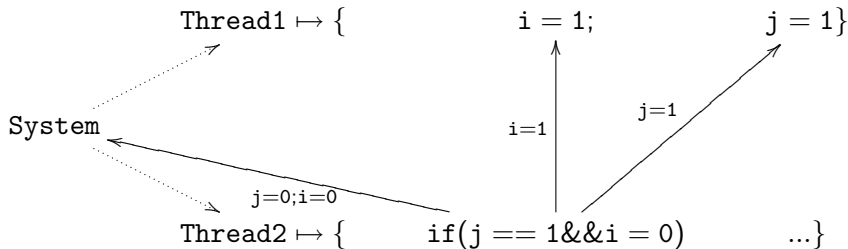
- ▶ We compute the semantics of three threads
- ▶ ... but the previous state exposes only the values written by System!

`visible(j) = {0}`

`visible(i) = {0}`

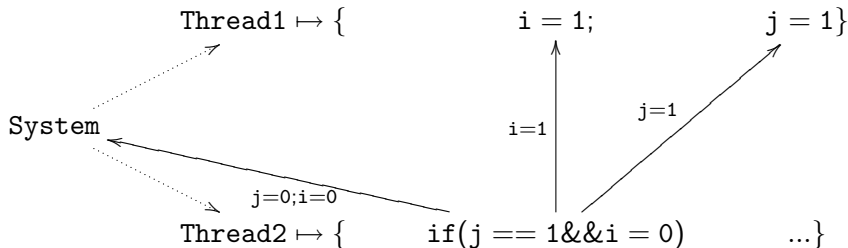
## An example - 3rd iteration

System	Thread 1	Thread 2
<code>new Thread1().start();</code>	<code>i = 1;</code>	<code>if(j == 1&amp;&amp; i == 0)</code>
<code>new Thread2().start();</code>	<code>j = 1;</code>	<code>throw new Exception();</code>



## An example - 3rd iteration

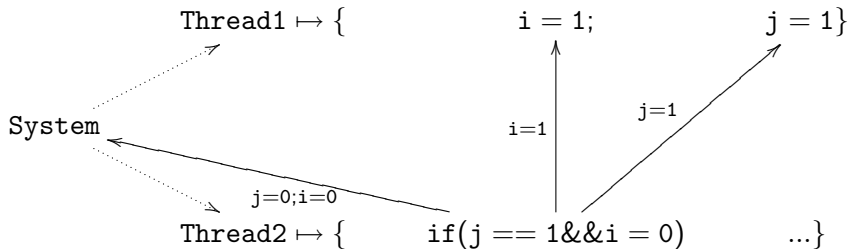
System	Thread 1	Thread 2
<code>new Thread1().start();</code>	<code>i = 1;</code>	<code>if(j == 1&amp;&amp; i == 0)</code>
<code>new Thread2().start();</code>	<code>j = 1;</code>	<code>throw new Exception();</code>



`visible(j) = {0, 1}`

## An example - 3rd iteration

System	Thread 1	Thread 2
<code>new Thread1().start();</code>	<code>i = 1;</code>	<code>if(j == 1&amp;&amp; i == 0)</code>
<code>new Thread2().start();</code>	<code>j = 1;</code>	<code>throw new Exception();</code>



`visible(j) = {0, 1}`

`visible(i) = {0, 1}`

# Outline

Memory Model

Concrete domain and semantics

**Abstraction**

Demo

Experimental results

Conclusion

- ▶ Exactly the same components of the concrete domain, but dealing with abstract elements
- ▶ Functions:
  - ▶  $shared^\# : \overline{St} \mapsto \overline{Sh}$
  - ▶  $action^\# : \overline{St} \mapsto \overline{\perp}_a \cup (\{\mathbf{r}, \mathbf{w}\} \times \overline{Loc} \times \overline{Val})$
  - ▶  $synchronised^\# : \overline{St} \mapsto \wp(\overline{Sync})$

[FER08] abstracts the concrete domain and intra-thread semantics

- ▶ Multithreaded state:

$$\begin{aligned}\overline{\Psi} : \overline{Tld} &\rightarrow \overline{St}^\dagger \\ \overline{\Omega} : \overline{Tld} &\rightarrow ((\overline{Tld} \times \overline{Integer}) \cup \overline{\perp}_\Omega)\end{aligned}$$

# Intra-thread semantics

- ▶  $visible^\#$  function: straight abstraction of the concrete one!
- ▶  $step^\#$  function: defines a single step in the computation.  
Makes the upper bound between all the visible values  $\rightarrow$  deterministic choice!

Definition (Single-thread semantics  $\bar{S}^\circ$ )

$$\begin{aligned} \bar{S}^\circ & : (\bar{\Psi} \times \bar{\Omega} \times \bar{TId}) \mapsto \bar{St}^\ddagger \\ \bar{S}^\circ \llbracket \bar{f}, \bar{r}, \bar{t} \rrbracket & = lfp_{\emptyset}^{\sqsubseteq} \lambda \bar{\tau}. \{ \bar{\sigma}_0 \} \sqcup_{\bar{\tau}} \{ \bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_{i-1} \rightarrow \bar{\sigma}_i : \\ & \quad \bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_{i-1} = \bar{\tau} \wedge \bar{\sigma}_i = step^\#(\bar{t}, \bar{f}, \bar{r}) \} \end{aligned}$$

## Definition (*visible*<sup>#</sup> function)

$$\begin{aligned}
 \text{visible}^\# & : \overline{TId} \times \overline{Loc} \times \wp(\overline{Sync}) \times \overline{\Psi} \times (\overline{TId} \times \text{Integer}) \mapsto \wp(\overline{Val}) \\
 \text{visible}^\# & (\bar{t}, \bar{l}, \bar{S}, \bar{f}, (\bar{t}', i')) = \\
 & = \text{project}^\#(\bar{l}, \text{suffix}^\#(\bar{f}(\bar{t}'), i'), \bar{S}) \cup \\
 & \{\bar{v} : \bar{v} \in \text{project}^\#(\bar{l}, \bar{f}(\bar{t}''), \bar{S}) : \bar{t}'' \in \text{dom}(\bar{f}) \setminus \{\bar{t}, \bar{t}'\}\}
 \end{aligned}$$

## Definition (*visible*<sup>#</sup> function)

$$\begin{aligned}
 \text{visible}^\# & : \overline{TId} \times \overline{Loc} \times \wp(\overline{Sync}) \times \overline{\Psi} \times (\overline{TId} \times \text{Integer}) \mapsto \wp(\overline{Val}) \\
 \text{visible}^\# & (\bar{t}, \bar{l}, \bar{S}, \bar{f}, (\bar{t}', i')) = \\
 & = \text{project}^\#(\bar{l}, \text{suffix}^\#(\bar{f}(\bar{t}'), i'), \bar{S}) \cup \\
 & \{ \bar{v} : \bar{v} \in \text{project}^\#(\bar{l}, \bar{f}(\bar{t}''), \bar{S}) : \bar{t}'' \in \text{dom}(\bar{f}) \setminus \{ \bar{t}, \bar{t}' \} \}
 \end{aligned}$$

## Definition (*visible* function)

$$\begin{aligned}
 \text{visible} & : TId \times Loc \times \wp(Sync) \times \Psi \times ((TId \times Integer) \cup \perp_\Omega) \mapsto \wp(Val) \\
 \text{visible}(t, l, S, f, (t', i')) & = \\
 (1) & = \text{project}(l, \text{suffix}(f(t'), i'), S) \cup \\
 (2) & \{ v : v \in \text{project}(l, f(t''), S) : t'' \in \text{dom}(f) \setminus \{ t, t' \} \}
 \end{aligned}$$

# Multithread semantics

- ▶ Each time we compute the single thread semantics of a thread it may expose new values to other threads
- ▶ Iterates the single-thread semantics until a fixpoint is reached
- ▶ ... as in the concrete semantics!

Definition (Multithreaded semantics  $\bar{S}^{\parallel}$ )

$$\bar{S}^{\parallel} : \bar{\Psi} \times \bar{\Omega} \mapsto \bar{\Psi} \times \bar{\Omega}$$

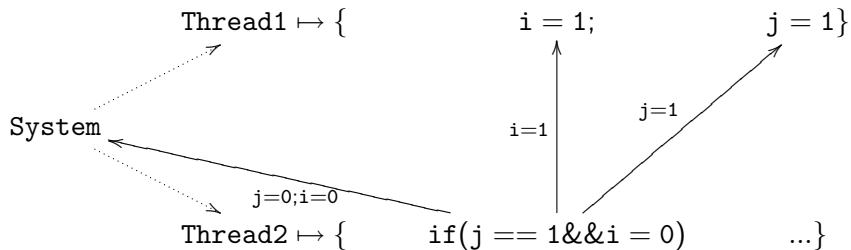
$$\bar{S}^{\parallel} \llbracket \bar{f}_0, \bar{r}_0 \rrbracket = \text{lfp}_{\emptyset}^{\sqsubseteq} \lambda(\bar{f}, \bar{r}). \{(\bar{f}_0, \bar{r}_0)\} \sqcup_f \{(\bar{f}_i, \bar{r}) : \forall \bar{t} \in \text{dom}(\bar{f}) : \bar{f}_i(\bar{t}) = \bar{S}^{\circ} \llbracket \bar{f}, \bar{t} \rrbracket\}$$

We formally proved the correctness of our analysis, i.e. :

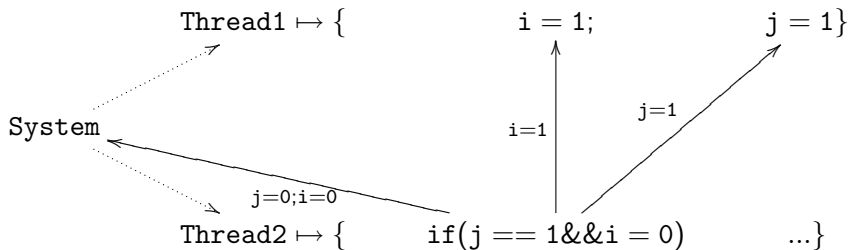
**Theorem (Soundness of  $\bar{S}^{\parallel}$ )**

$$\forall \bar{f} \in \bar{\Psi}_{pre} : \alpha_f(S^{\parallel})[[\bar{f}]] \sqsubseteq_f \bar{S}^{\parallel}[[\bar{f}]].$$

## An example - 3rd iteration

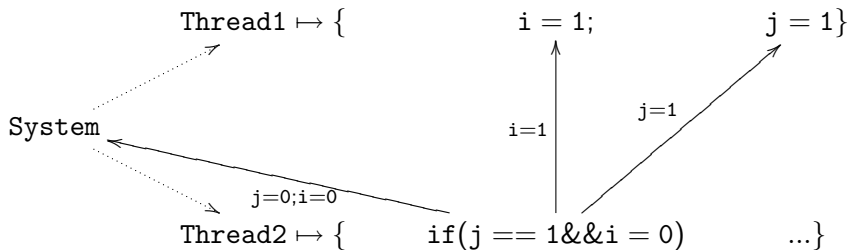


## An example - 3rd iteration



$$\text{visible}^\#(j) = [0..0] \sqcup [1..1] = [0..1]$$

## An example - 3rd iteration



$$\text{visible}^\#(j) = [0..0] \sqcup [1..1] = [0..1]$$

$$\text{visible}^\#(i) = [0..0] \sqcup [1..1] = [0..1]$$

# Outline

Memory Model

Concrete domain and semantics

Abstraction

**Demo**

Experimental results

Conclusion

# Outline

Memory Model

Concrete domain and semantics

Abstraction

Demo

Experimental results

Conclusion

# Implementation

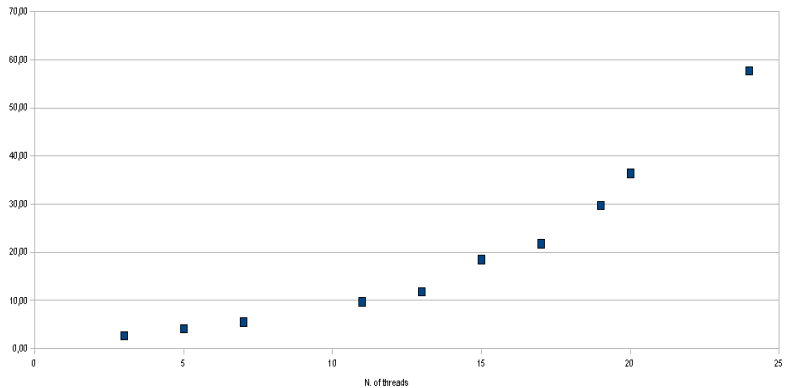
- ▶ We have implemented the analysis
  - ▶ analyze (a subset of) Java bytecode
  - ▶ developed in Java
- ▶ Tested on a set of 10 applications built ad hoc
- ▶ Executed on an AMD Athlon 64 3000+ processor with 2 GB of RAM running a GNU/Linux operating system with the Java virtual machine version 1.5.0
- ▶ The analysis has been executed 10 times on each application and we take the average on the times of execution

# Results

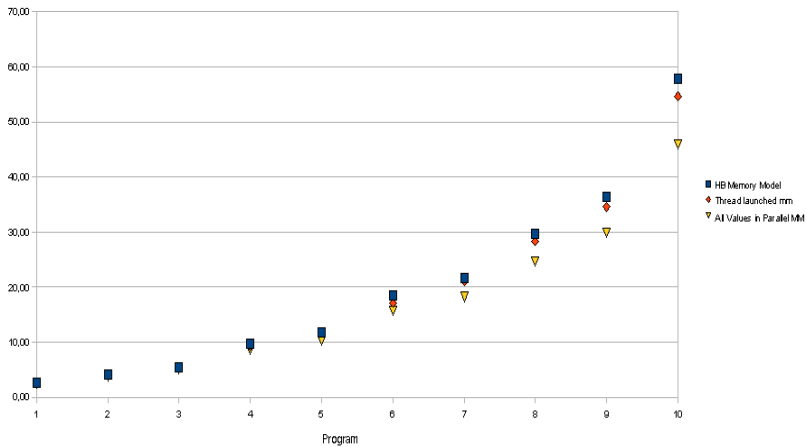
# Pr.	# Th.	# St.	HB (sec)	TS (sec)	AP (sec)
1	3	424	2,63	2,58	2,51
2	5	637	4,06	4,03	3,81
3	7	751	5,44	5,32	4,99
4	11	977	9,70	9,41	8,53
5	13	1.097	11,83	11,83	10,26
6	15	1.310	18,47	17,14	15,77
7	17	1.422	21,75	21,11	18,22
8	19	1.635	29,70	28,30	24,67
9	20	1.742	36,37	34,54	29,86
10	24	2.126	57,76	54,62	45,98

Table: Experimental results

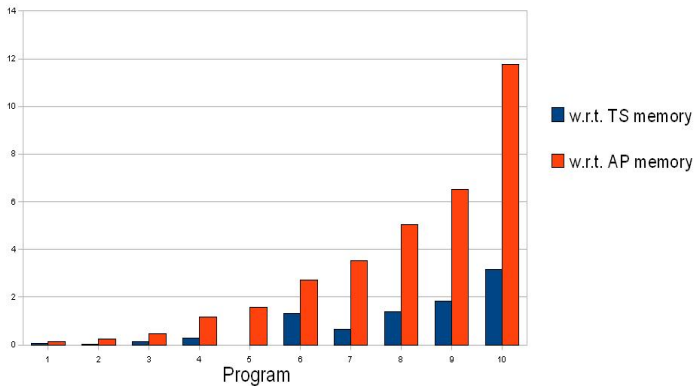
Execution time vs N. of threads  
using HB memory model and top domain



Memory models  
using top domain



## Overhead of the Happens-Before memory model



# Outline

Memory Model

Concrete domain and semantics

Abstraction

Demo

Experimental results

**Conclusion**

- ▶ Many papers have been focused on the analysis of specific properties like data race condition [RIN01]. Usually these approaches suppose that the executions are sequentially consistent
- ▶ Other papers define memory models [MAN05,CEN07,SAR07] but they do not propose any static analysis
- ▶ Two model checkers [ROY02,HUY06] sound w.r.t. a memory model have been proposed in the last years. They are afflicted by the state explosion problem, and they seem not in position to scale up

# Conclusion

- ▶ We formalize in fixpoint form the happens-before memory model
  - ▶ generic w.r.t. the programming language
  - ▶ supports mutual exclusion
- ▶ We abstract it
  - ▶ The main characteristic components of the memory model are straightly abstracted!
    - ▶ For the next memory model, define the concrete semantics and obtain the abstract one!
- ▶ We prove formally the soundness of our abstraction
- ▶ We implement it and the preliminary experimental results are quite encouraging
- ▶ As far as we know, it is the first static analysis sound w.r.t. the happens-before memory model

## Bibliography - Memory Models and Concurrency

**[LAM79]** Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. In IEEE Trans. Computers, volume 28, pages 690691, 1979.

**[LAM78]** Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In Commun. ACM, volume 21, pages 558565, New York, NY, USA, 1978. ACM Press.

**[MAN05]** J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In ACM Press, editor, Proceedings of POPL 05, 2005.

**[LEE06]** Edward A. Lee. The problem with threads. In IEEE Computer, volume 39, pages 3342, Los Alamitos, CA, USA, May 2006. IEEE Computer Society Press.

**[SUT05]** Herb Sutter and James Larus. Software and the concurrency revolution. In ACM Queue, Vol. 3, No. 7, pages 5462, September 2005.

## Bibliography - Related Works

**[RIN01]** Martin C. Rinard. Analysis of multithreaded programs. In SAS 01, pages 119, London, UK, 2001. Springer-Verlag.

**[CEN07]** Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The java memory model: Operationally, denotationally, axiomatically. In Springer Verlag, editor, Proceeding of ESOP 07, number 4421 in Lecture Notes in Computer Science, pages 331346, 2007.

**[SAR07]** Vijay Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. A theory of memory models. To appear (<http://www.saraswat.org/raofull.pdf>), 2006.

**[ROY02]** Abhik Roychoudhury and Tulika Mitra. Specifying multithreaded java semantics for program verification. In ACM Press, editor, Proceedings of ICSE, May 2002.

**[HUY06]** Thuan Quang Huynh and Abhik Roychoudhury. A memory model sensitive checker for c#. In Springer-Verlag, editor, FM, volume 4085 of Lecture Notes in Computer Science, 2006.