Simple thread semantics require race detection

Hans-J. Boehm
Why data race detectors?

• Debugging
  – Code with data races is
    • Suspect.
      – VERY hard to analyze correctness (stay tuned).
    • Probably wrong.
    • Definitely wrong in many environments (e.g. C++0x, C1x, Ada, C + Posix)
    • Hard to debug at point of failure.
  – But you knew that already.
Why else data race detectors?

• Easy to specify semantics of programs without data races (cf. work by S. Adve).
  – Conflicting accesses must be ordered by synchronization (happens before).
  – Each load “sees” the unique store that
    • happens before the load, and
    • happens after all other such stores to that location.

\[
\begin{align*}
\text{T1} & : & x = 1 & \text{synchronize} \\
\text{T2} & : & x = 2 & \text{synchronize} \\
\text{T3} & : & r1 = x
\end{align*}
\]
What about programs with data races?
Interesting data race outcome 1

\( x \) initially zero

---

Thread 1:
\[ x = 100000; \]

Thread 2:
\[ x = 40000; \]

Outcome: \( x = 105536 \)
(But Java has a hack ...)
Interesting data race outcome?

\[x, \ y\] initially null,

Loads may or may not see racing stores?

**Thread 1:**
\[
\begin{align*}
\text{r1} &= \text{x}; \\
\text{y} &= \text{r1};
\end{align*}
\]

**Thread 2:**
\[
\begin{align*}
\text{r2} &= \text{y}; \\
\text{x} &= \text{r2};
\end{align*}
\]

Outcome: \[x = y = r1 = r2 = \text{"<your bank password here>"}\]
Standard solutions

• Undefined semantics for data races
  – Ada, Posix, C++0x, C1x
  – Doesn’t address security issues
  – Unacceptable for Java

• Statically prevent data races
  – generality vs. type system complexity
  – sometimes appropriate, but not widely used

• Java causality treatment
The Java Solution

Quotation from 17.4.8, Java Language Specification, 3rd edition, omitted, to avoid possible copyright questions. The important point is that this is a rather complex mathematical specification.
Complicated, but nice properties?

• Manson, Pugh, Adve: The Java Memory Model, POPL 05

Quotation from section 9.1.2 of above paper omitted, to avoid possible copyright questions. This asserts (Theorem 1) that non-conflicting operations may be reordered by a compiler.
Much nicer than prior attempts, but:

- Aspinall, Sevcik, “Java Memory Model Examples: Good, Bad, and Ugly”, VAMP 2007 (also ECOOP 2008 paper)

Quotation from above paper omitted, to avoid possible copyright questions. This ends in the statement:

“This falsifies Theorem 1 of [paper from previous slide].”

Note 1: This does not necessarily mean implementations are broken, or that we know how to do better. It does suggest this is too complicated.

Note 2: The underlying observation is due to Pietro Cenciarelli.
Another way out

• Accurate data-race detectors allow us to avoid the issue, e.g.:
  – Goldilocks (Elmas et al, PLDI ‘07), or
  – FastTrack (Flanagan and Freund, PLDI ‘09)

• All loads either
  – See a store that “happens before” it, or
  – Raise an exception.

• Causal cycles can’t arise.

• Independence from access granularity.

• No security issues.
What about cost?

• Definitely still an issue 😞
  – No sampling allowed!
• No need to detect write-after-read races, the most expensive kind 😊
• We don’t have to detect exactly data races (e.g. Ceze et al, HotPar 09) …

• Neither clearly solvable, nor clearly a show-stopper?
Questions?