C++ Concurrency - Formalised

Salomon Sickert

Technische Universität München

26^{th} April 2013

Mutex Algorithms

- At most one thread is in the critical section at any time.

Dekker's Mutex Algorithm [2]

Initialisation

```
x = 0; y = 0;
```

Thread 1

```
x = 1;
if (y == 1) {
  ... //Busy Wait
}
// Critical Section
```

Thread 2

```
y = 1;
if (x == 1) {
  ... //Busy Wait
}
// Critical section
```

Dekker's Mutex Algorithm [2]

Initialisation

```
x = 0; y = 0;
```

Thread 1

```
x = 1;
if (y == 1) {
  ... //Busy Wait
}
// Critical Section
```

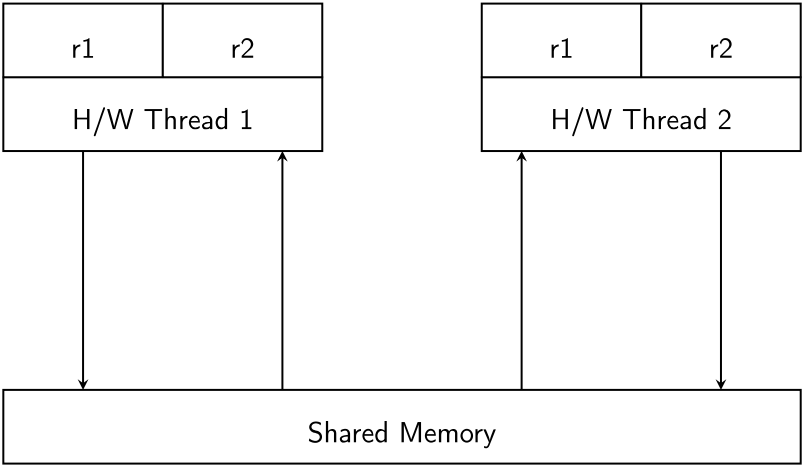
```
MOV [x] <- 1
MOV r1 <- [y]
```

Thread 2

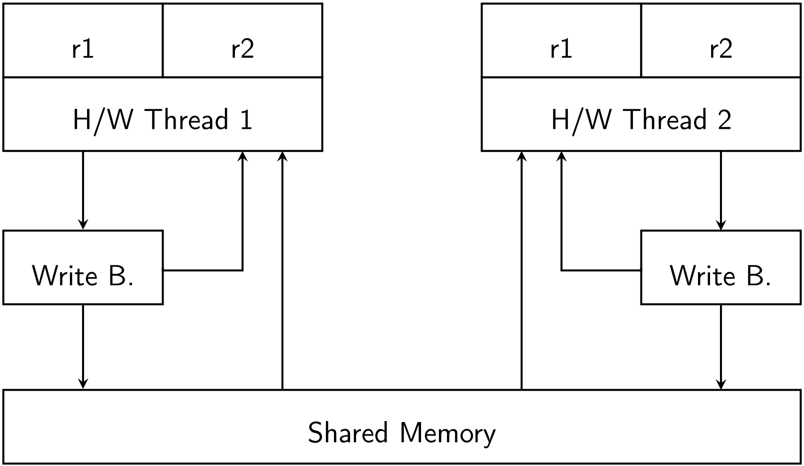
```
y = 1;
if (x == 1) {
  ... //Busy Wait
}
// Critical section
```

```
MOV [y] <- 1
MOV r2 <- [x]
```

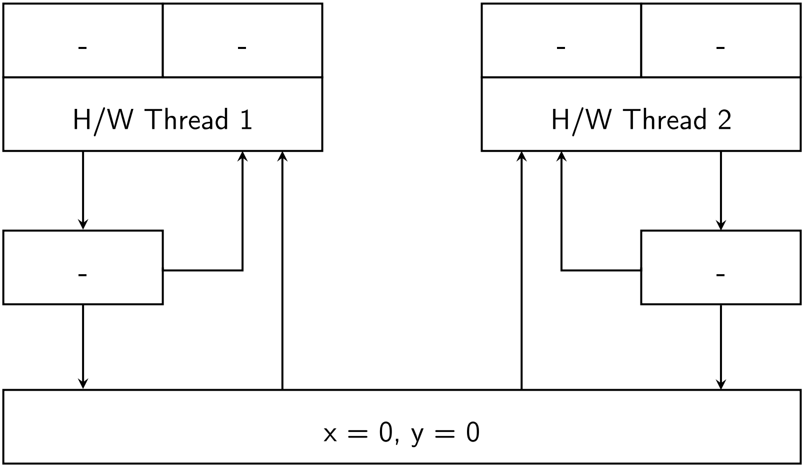
Modern x86 Multiprocessors - simplified (based on [4])



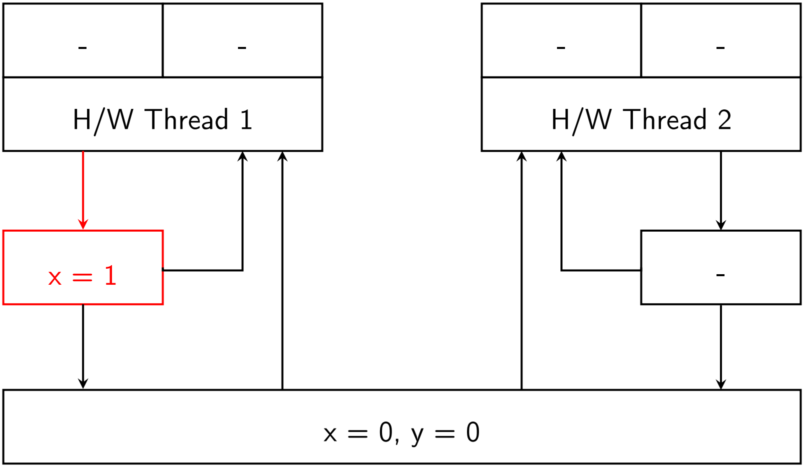
Modern x86 Multiprocessors - simplified



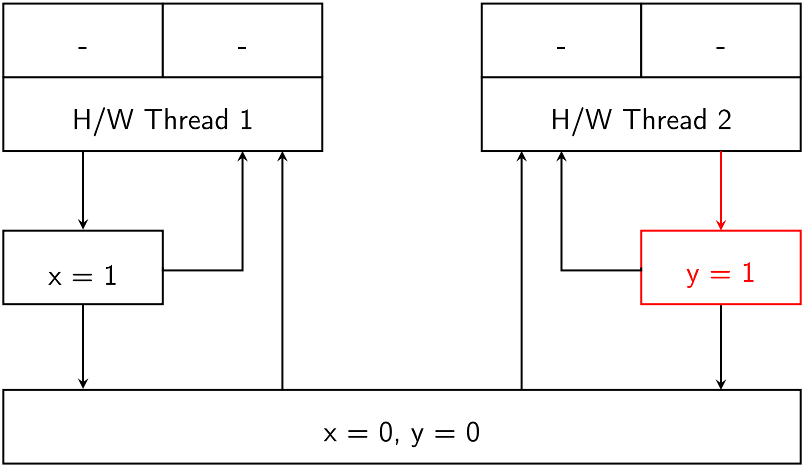
Modern x86 Multiprocessors and Dekker's Algorithm



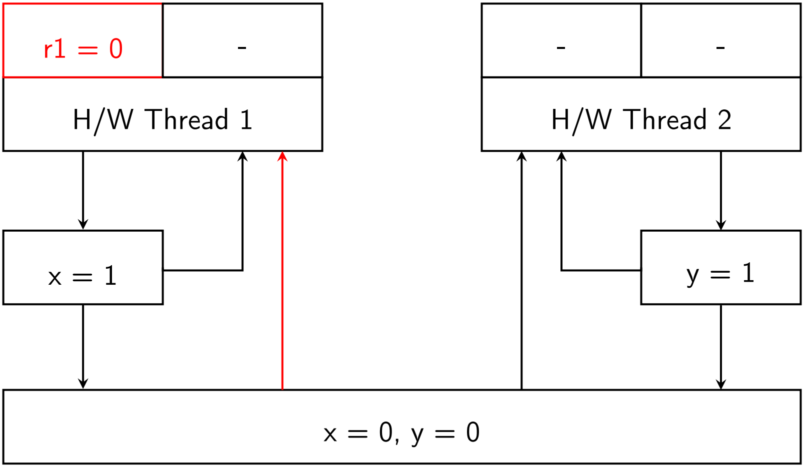
Modern x86 Multiprocessors and Dekker's Algorithm



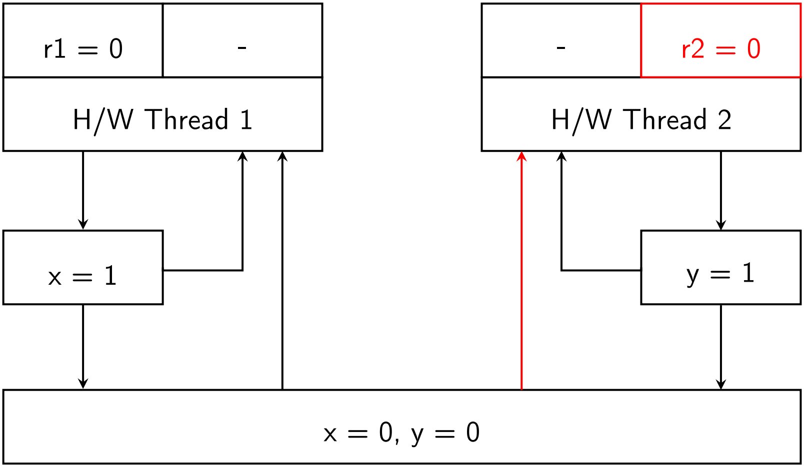
Modern x86 Multiprocessors and Dekker's Algorithm



Modern x86 Multiprocessors and Dekker's Algorithm



Modern x86 Multiprocessors and Dekker's Algorithm



Modern x86 Multiprocessors

- Both threads may enter the critical section at the **same time**!

Modern x86 Multiprocessors

- Both threads may enter the critical section at the **same time**!
- Due to write buffers threads may see a **subtly different memory**.

Modern x86 Multiprocessors

- Both threads may enter the critical section at the **same time**!
- Due to write buffers threads may see a **subtly different memory**.
- The processor exhibits a *relaxed memory model*.

Modern x86 Multiprocessors

- Both threads may enter the critical section at the **same time**!
- Due to write buffers threads may see a **subtly different memory**.
- The processor exhibits a *relaxed memory model*.
- Solutions:
  - Assembler: FENCE instructions.
  - C++11: Special types. (Atomics)

Modern x86 Multiprocessors

- Both threads may enter the critical section at the **same time**!
- Due to write buffers threads may see a **subtly different memory**.
- The processor exhibits a *relaxed memory model*.
- Solutions:
  - Assembler: FENCE instructions.
  - C++11: Special types. (Atomics)
- Sequential consistency is restored by paying a performance penalty.

Modern x86 Multiprocessors

- Both threads may enter the critical section at the **same time**!
- Due to write buffers threads may see a **subtly different memory**.
- The processor exhibits a *relaxed memory model*.
- Solutions:
  - Assembler: FENCE instructions.
  - C++11: Special types. (Atomics)
- Sequential consistency is restored by paying a performance penalty.
- Some Non-x86 architectures exhibit even weaker models, e.g ARM.

Mathematizing C++ Concurrency ([3])

- Given a program p, what are the possible ways to execute it?

Mathematatizing C++ Concurrency ([3])

- Given a program p, what are the possible ways to execute it?
  1. Calculate X_{opsem} from p.
     - Loosely speaking: Structure of the program.

Mathematizing C++ Concurrency ([3])

- Given a program p, what are the possible ways to execute it?
  1. Calculate X_{opsem} from p.
     - Loosely speaking: Structure of the program.
  2. Find all $X_{witness}$ consistent with X_{opsem}.
     - Loosely speaking: Different executions of the program.

Mathematizing C++ Concurrency ([3])

- Given a program p, what are the possible ways to execute it?
  1. Calculate X_{opsem} from p.
     - Loosely speaking: Structure of the program.
  2. Find all $X_{witness}$ consistent with X_{opsem}.
     - Loosely speaking: Different executions of the program.
  3. Check for undefined behaviour.
     - Reading from uninitialized variables
     - Unsequenced Races (e.g. x == (x = 2))
     - Data Races
     - ...

X_{opsem}: An Example

```
int main() {
  int x, y, z, a;
  {{{  x = 1;
  |||  y = 2;
  }}};
  a = 1;
  z = (x == y);
  return 0;
}
```

X_{opsem}: An Example

```
int main() {
  int x, y, z, a;
  {{{  x = 1;
  |||  y = 2;
  }}};
  a = 1;
  z = (x == y);
  return 0;
}
```

e:W x=1

X_{opsem}: An Example

```
int main() {
  int x, y, z, a;
  {{{  x = 1;
  |||  y = 2;
  }}};
  a = 1;
  z = (x == y);
  return 0;
}
```

f:W y=2

e:W x=1

X_{opsem}: An Example

```c
int main() {
  int x, y, z, a;
  {{{  x = 1;
  |||  y = 2;
  }}};
  a = 1;
  z = (x == y);
  return 0;
}
```

f:W y=2

e:W x=1

a:W a=1

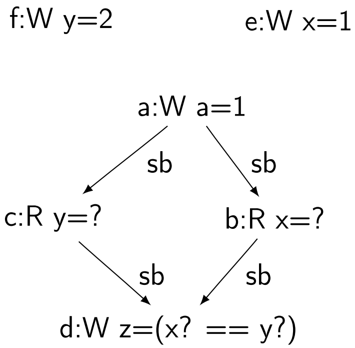
c:R y=?

b:R x=?

d:W z=(x? == y?)

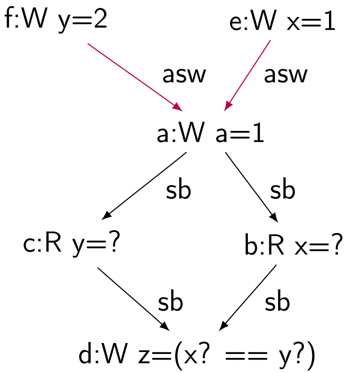
X_{opsem}: An Example

```
int main() {
  int x, y, z, a;
  {{{  x = 1;
  |||  y = 2;
  }}};
  a = 1;
  z = (x == y);
  return 0;
}
```



X_{opsem}: An Example

```
int main() {
  int x, y, z, a;
  {{{  x = 1;
  |||  y = 2;
  }}};
  a = 1;
  z = (x == y);
  return 0;
}
```



X_{opsem}

- Independent from the architecture

X_{opsem}

- Independent from the architecture
- Composed of (among other parts):
  - Actions (simplified)
    - $aid : R\ l = v$
    - $aid : W\ l = v$

X_{opsem}

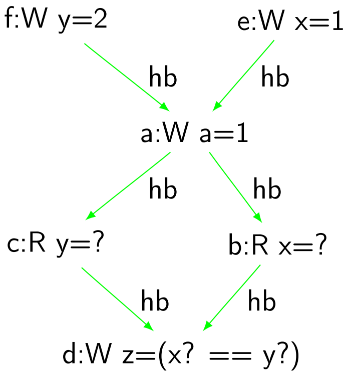
- Independent from the architecture
- Composed of (among other parts):
  - Actions (simplified)
    - $aid : R\ l = v$
    - $aid : W\ l = v$
  - Binary relations:
    - $\xrightarrow{sequenced-before}$ (sb)
    - $\xrightarrow{additional-synchronized-with}$ (asw)

X_{opsem}

- Independent from the architecture
- Composed of (among other parts):
  - Actions (simplified)
    - $aid : R\ l = v$
    - $aid : W\ l = v$
  - Binary relations:
    - $\xrightarrow{sequenced-before}$ (sb)
    - $\xrightarrow{additional-synchronized-with}$ (asw)
- In this special case: $\xrightarrow{simple-happens-before} = (\xrightarrow{sequenced-before} \cup \xrightarrow{additional-synchronized-with})^+$

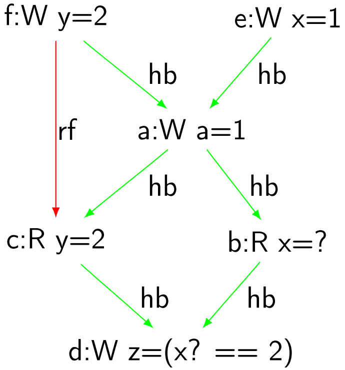
$X_{witness}$: An Example

```
int main() {
  int x, y, z, a;
  {{{  x = 1;
  |||  y = 2;
  }}};
  a = 1;
  z = (x == y);
  return 0;
}
```



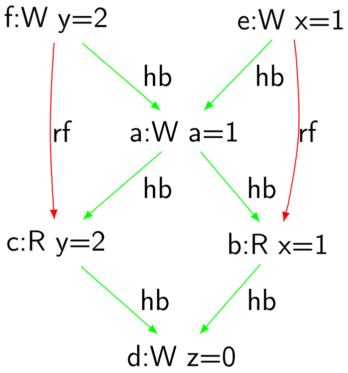
$X_{witness}$: An Example

```
int main() {
  int x, y, z, a;
  {{{  x = 1;
  |||  y = 2;
  }}};
  a = 1;
  z = (x == y);
  return 0;
}
```



$X_{witness}$: An Example

```
int main() {
  int x, y, z, a;
  {{{  x = 1;
  |||  y = 2;
  }}};
  a = 1;
  z = (x == y);
  return 0;
}
```



$X_{witness}$

- Dependent on the architecture

$X_{witness}$

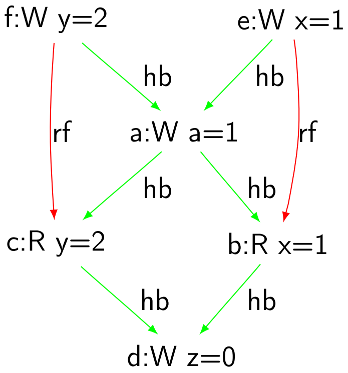
- Dependent on the architecture
- Composed of:
  - Binary relations:
    - $\xrightarrow{reads-from}$ (rf)

$X_{witness}$

- Dependent on the architecture
- Composed of:
  - Binary relations:
    - $\xrightarrow{reads-from}$ (rf)
    - $\xrightarrow{sequentialconsistency}$ (sc) (not applicable in the example)
    - $\xrightarrow{modificationorder}$ (mo) (not applicable in the example)

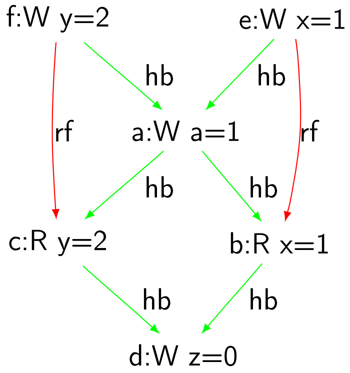
$X = (X_{opsem}, X_{witness})$: An execution candidate

```
int main() {
  int x, y, z, a;
  {{{  x = 1;
  |||  y = 2;
  }}};
  a = 1;
  z = (x == y);
  return 0;
}
```



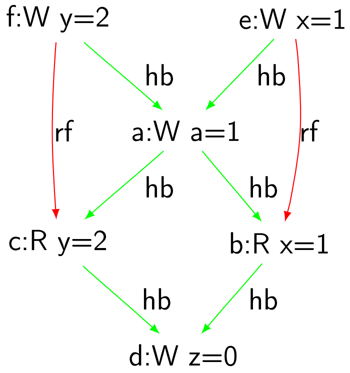
$X = (X_{opsem}, X_{witness})$: Undefined Behaviour?

- Uninitialised Reads?
- Unsequenced Races?
- Data Races?



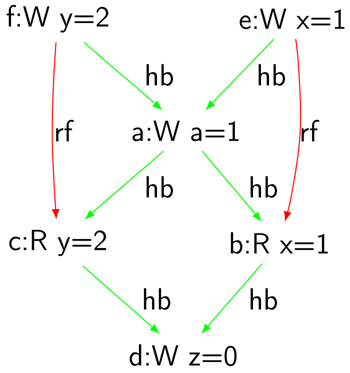
$X = (X_{opsem}, X_{witness})$: Undefined Behaviour?

- Uninitialised Reads? $\times$
- Unsequenced Races?
- Data Races?



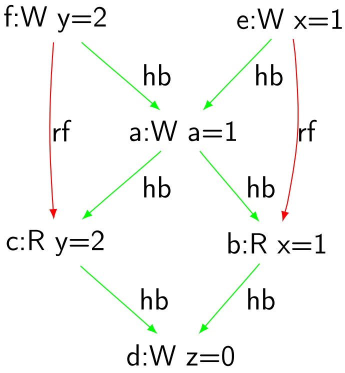
$X = (X_{opsem}, X_{witness})$: Undefined Behaviour?

- Uninitialised Reads? $\times$
- Unsequenced Races? $\times$
- Data Races?



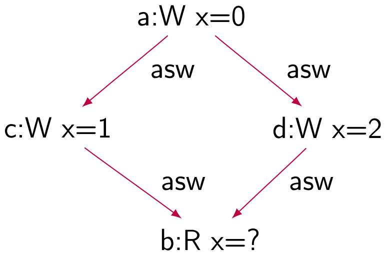
$X = (X_{opsem}, X_{witness})$: Undefined Behaviour?

- Uninitialised Reads? $\times$
- Unsequenced Races? $\times$
- Data Races? $\times$



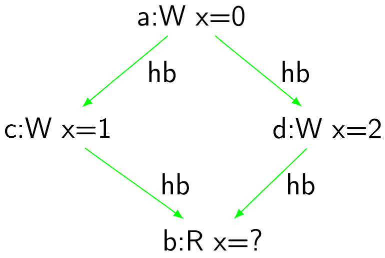
Undefined Behaviour: Data Races

```
int main() {
  int x = 0;
  {{{ x = 1;
  ||| x = 2;
  }}};
  return x;
}
```



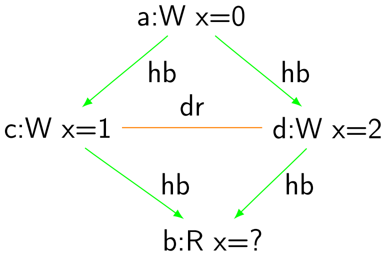
Undefined Behaviour: Data Races

```
int main() {
  int x = 0;
  {{{ x = 1;
  ||| x = 2;
  }}};
  return x;
}
```



Undefined Behaviour: Data Races

```
int main() {
  int x = 0;
  {{{ x = 1;
  ||| x = 2;
  }}};
  return x;
}
```



The Formalised C++ Memory Model

$$
\begin{array}{l}
\text{cpp\_memory\_model } \textit{opsem } (p : \text{program}) = \\
\quad \textbf{let } \text{pre\_executions} = \{(X_{opsem}, X_{witness}) \mid \\
\qquad \textit{opsem } p\ X_{opsem} \wedge \text{consistent\_execution } X_{opsem}\ X_{witness}\} \textbf{ in} \\
\quad \textbf{if } \exists X \in \text{pre\_executions}.\ \text{undefined\_behaviour } X \\
\quad \textbf{then } \text{NONE} \\
\quad \textbf{else } \text{SOME pre\_executions}
\end{array}
$$

C++11 Concurrency / Language Features

Concurrency Idioms (**Atomics**, Mutexes, Threads)

C++11 Concurrency / Language Features

Concurrency Idioms (**Atomics**, Mutexes, Threads)

Pre-C++11

- No memory model for multi-threaded code
- Concurrency Idioms provided by a third party:
  - pthreads
  - OpenMP
- Drawbacks: No formalised standard, compiler may produce incorrect code.

C++11 Concurrency / Language Features

Concurrency Idioms (**Atomics**, Mutexes, Threads)

Pre-C++11

- No memory model for multi-threaded code
- Concurrency Idioms provided by a third party:
  - pthreads
  - OpenMP
- Drawbacks: No formalised standard, compiler may produce incorrect code.

C++11

- A memory model for multi-threaded code
- Concurrency Idioms in are part of the language (std::atomic<T> [1], std::mutex, std::thread)
  - Similar to Java
- Benefits: Compiler is able to produce correct code.

Applications of the Formal Memory Model

- Corrections to the C++0x standard.
  - memory_order_seq_cst was in fact not sequentially consistent

Applications of the Formal Memory Model

- Corrections to the C++0x standard.
  - memory_order_seq_cst was in fact not sequentially consistent
- Confidence in memory model and specification.

Applications of the Formal Memory Model

- Corrections to the C++0x standard.
  - memory_order_seq_cst was in fact not sequentially consistent
- Confidence in memory model and specification.
- Verify correctness of prototype implementations.

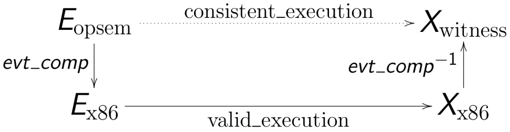


Figure: Figure taken from [3]

Applications of the Formal Memory Model

- Corrections to the C++0x standard.
  - memory_order_seq_cst was in fact not sequentially consistent
- Confidence in memory model and specification.
- Verify correctness of prototype implementations.

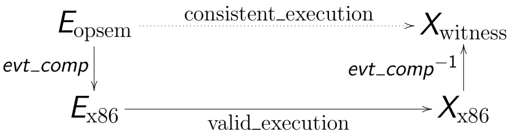


Figure: Figure taken from [3]

- Developer support.

Questions?

Bonus: Atomics vs. Mutexes - short presentation

salomon@sputnik:~/Documents/Studium/Semester 8/Master Seminar Software Verification/code

File Edit View Search Terminal Help

```
[salomon@sputnik code]$ ./non_atomic.o
Final counter value is 8785553
Elapsed Time is 70000
[salomon@sputnik code]$ ./atomic.o
Final counter value is 10000000
Elapsed Time is 370000
[salomon@sputnik code]$ ./mutex.o
Final counter value is 10000000
Elapsed Time is 3.47e+06
[salomon@sputnik code]$
```

Bonus: Dekker's algorithm in CppMem

http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/index.html

References

- C++ atomic types and operations, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2427.html.
- Dekker's algorithm, http://www.cs.utexas.edu/users/ewd/transcriptions/ewd01xx/ewd123.
- Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber.
  Mathematizing c++ concurrency.
  SIGPLAN Not., 46(1):55–66, January 2011.
- Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen.
  x86-tso: a rigorous and usable programmer's model for x86 multiprocessors.
  Commun. ACM, 53(7):89–97, July 2010.