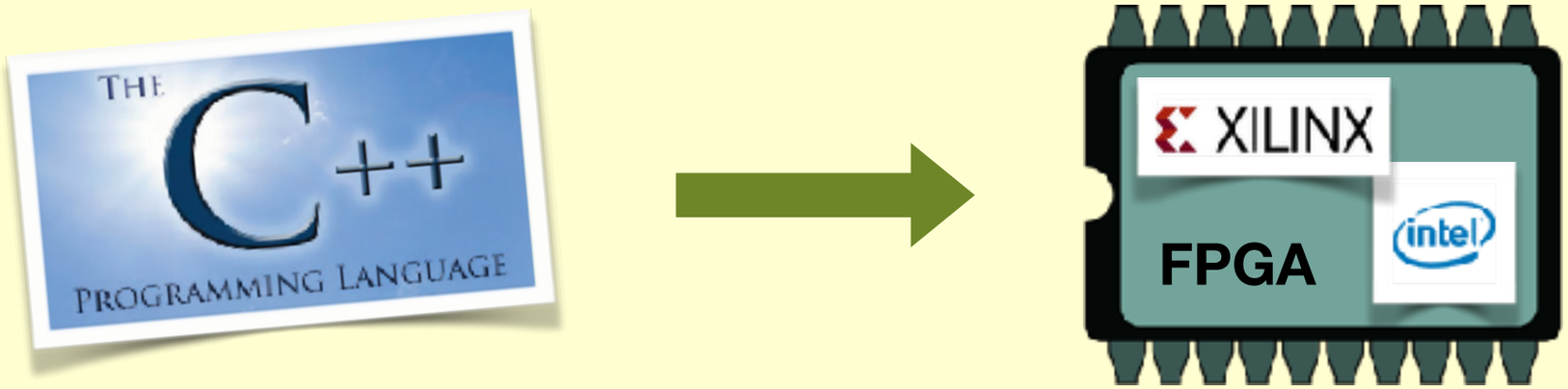


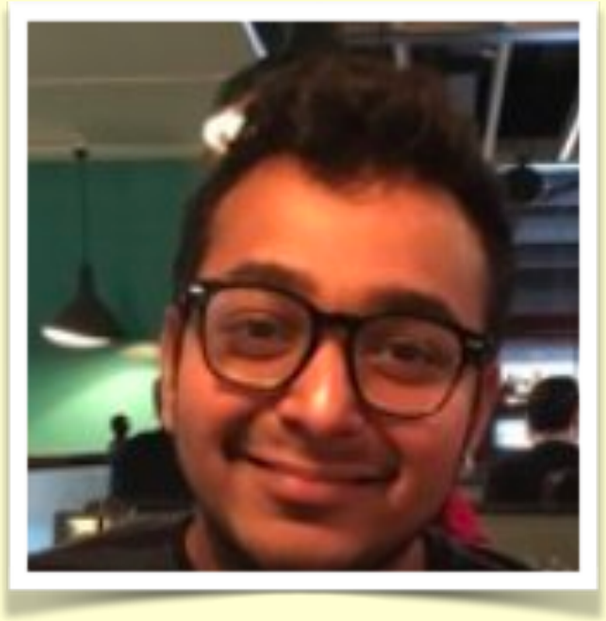
Towards Verified Hardware Compilation

John Wickerson

Imperial College London

FMATS Workshop, Microsoft Research Cambridge, 24 Sep 2018

Collaborators



Nadesh Ramanathan

George Constantinides

Hardware Compilation?

Hardware Compilation?

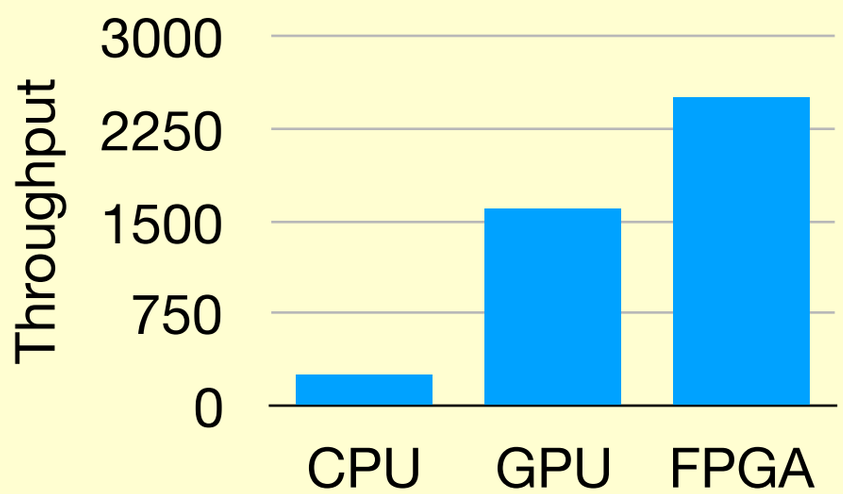
- Also called "high-level synthesis".

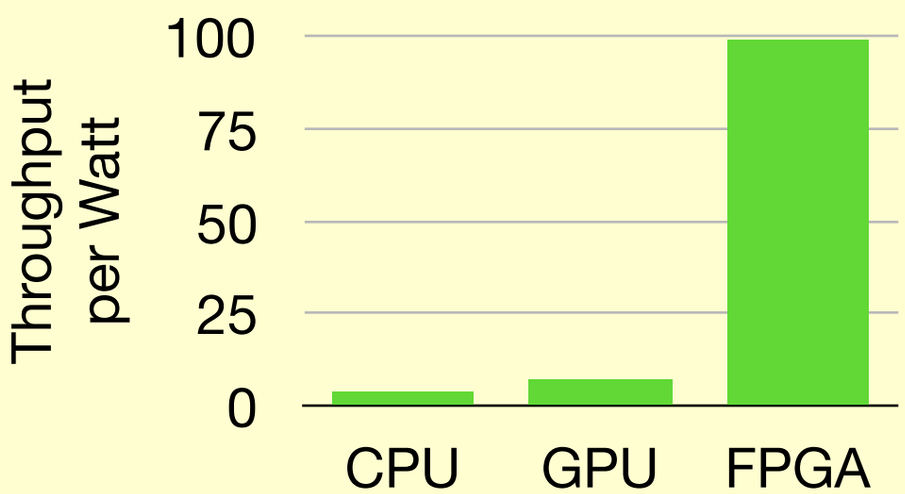
Hardware Compilation?

- Also called "high-level synthesis".
- Basic idea: translate C (or OpenCL, or ...) to Verilog.

Hardware Compilation?

- Also called "high-level synthesis".
- Basic idea: translate C (or OpenCL, or ...) to Verilog.
- Custom hardware can be 10x faster and 10x more power-efficient than running software on a processor.





(1) S.O. Settle, "High-performance Dynamic Programming on FPGAs with OpenCL", in *High Performance Extreme Computing (HPEC)*, 2013.

Hardware Compilation?

Hardware Compilation?

- Use of hardware compilers has grown ~20x since 2011.[2]

(2) S. Raje, "Extending the power of FPGAs to software developers", in *Field-Programmable Logic and Applications (FPL)*, 2015. Keynote.

Hardware Compilation?

- Use of hardware compilers has grown ~20x since 2011.(2)
- There are ~19x more software engineers than hardware engineers.(3)

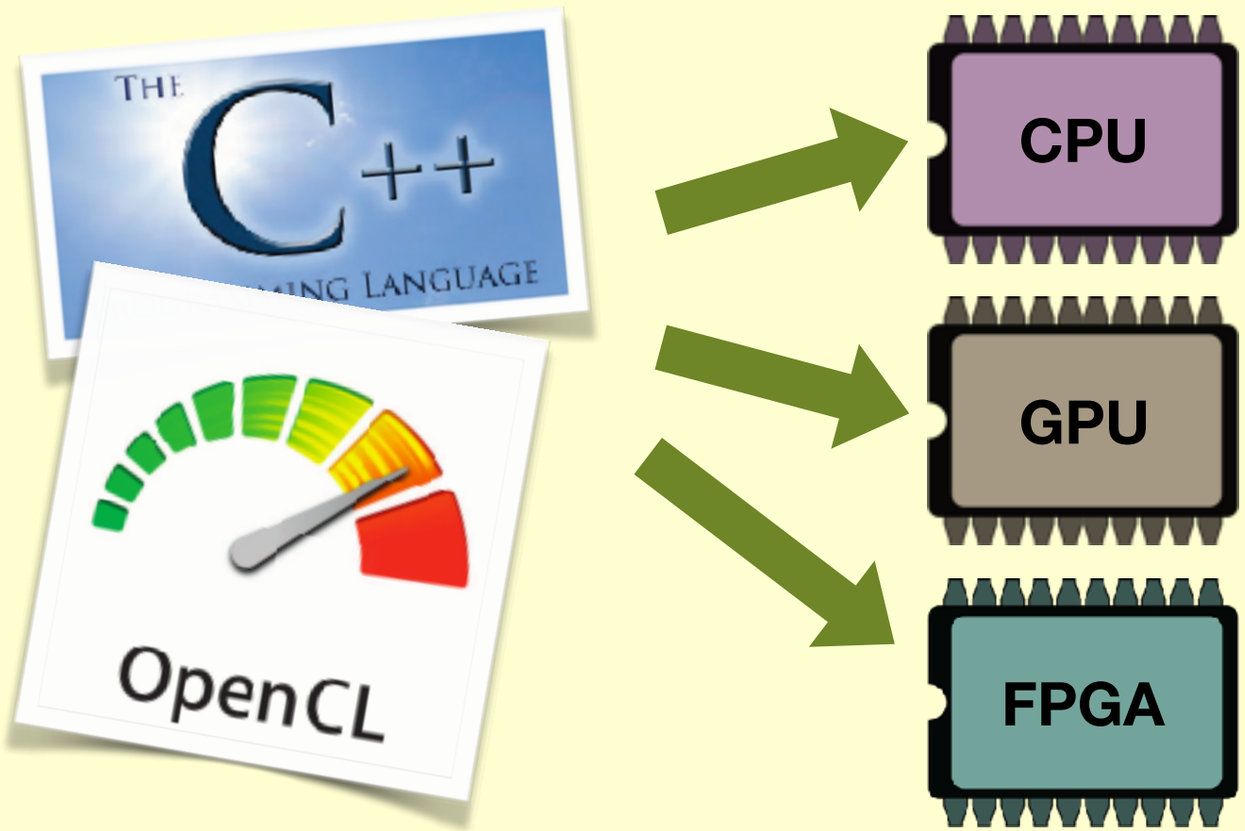
(2) S. Raje, "Extending the power of FPGAs to software developers", in *Field-Programmable Logic and Applications (FPL)*, 2015. Keynote.
(3) United States Bureau of Labor Statistics, "Occupational Outlook Handbook, 2016–17 Edition", 2015.

Hardware Compilation?

- Use of hardware compilers has grown ~20x since 2011.(2)
- There are ~19x more software engineers than hardware engineers.(3)
- A user survey found "Lack of C-to-RTL formal verification" to be the biggest problem with hardware compilation.(4)

(2) S. Raje, "Extending the power of FPGAs to software developers", in *Field-Programmable Logic and Applications (FPL)*, 2015. Keynote.
(3) United States Bureau of Labor Statistics, "Occupational Outlook Handbook, 2016–17 Edition", 2015.
(4) Deep Chip, “Survey on HLS verification issues and power reduction”, 2014. http://www.deepchip.com/items/0544-03.html

Hardware Compilation of Concurrency



Atomic operations

Atomic operations

- Atomics must appear to execute **instantaneously** to other threads

Atomic operations

- Atomics must appear to execute **instantaneously** to other threads
- Atomics provide a variety of **ordering** guarantees

Atomic operations

- Atomics must appear to execute **instantaneously** to other threads

- Atomics provide a variety of **ordering** guarantees

```
x = 1;                          ||  r = atomic_load(&y,
atomic_store(&y, 1,             ||    memory_order_acquire);
  memory_order_release);        ||  if (r==1) { print(x); }
```

Atomic operations

- Atomics must appear to execute **instantaneously** to other threads

- Atomics provide a variety of **ordering** guarantees

```
x = 1;                          ||  r = atomic_load(&y,
atomic_store(&y, 1,             ||    memory_order_acquire);
  memory_order_release);        ||  if (r==1) { print(x); }
```

```
r1 = atomic_load(&x,            ||  atomic_store(&x, 1,
  memory_order_relaxed);        ||    memory_order_relaxed);
r2 = atomic_load(&x,            ||
  memory_order_relaxed);        ||
```

Weak-memory concurrency is tricky!

Weak-memory concurrency is tricky!

- x86 proved tricky to formalise correctly.[5,6]

(5) Sarkar et al., *POPL*, 2009.
(6) Owens et al., *TPHOLs*, 2009.

Weak-memory concurrency is tricky!

- x86 proved tricky to formalise correctly.[5,6]
- Bug found in deployed "IBM Power 5" processors.[7]

(5) Sarkar et al., *POPL*, 2009.
(6) Owens et al., *TPHOLs*, 2009.
(7) Alglave et al., *CAV*, 2010.

Weak-memory concurrency is tricky!

- x86 proved tricky to formalise correctly.[5,6]
- Bug found in deployed "IBM Power 5" processors.[7]
- C++ specification did not guarantee its own key property.[8]

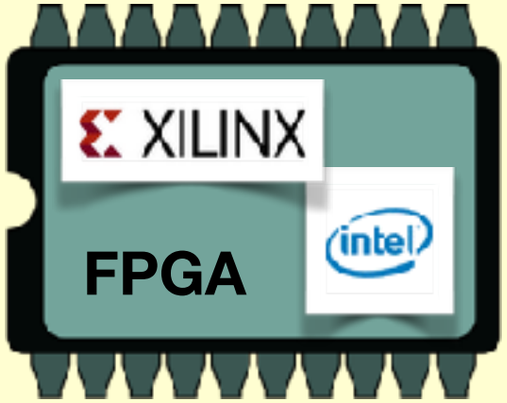
(5) Sarkar et al., *POPL*, 2009.
(6) Owens et al., *TPHOLs*, 2009.
(7) Alglave et al., *CAV*, 2010.
(8) Batty et al., *POPL*, 2011.

Weak-memory concurrency is tricky!

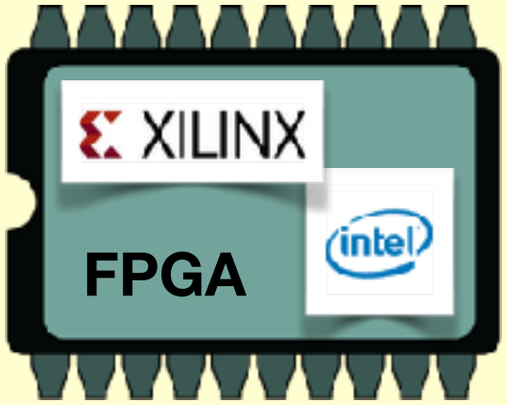
- x86 proved tricky to formalise correctly.[5,6]
- Bug found in deployed "IBM Power 5" processors.[7]
- C++ specification did not guarantee its own key property.[8]
- Behaviour of NVIDIA's graphics processors contradicted their own programming guide.[9]

(5) Sarkar et al., *POPL*, 2009.
(6) Owens et al., *TPHOLs*, 2009.
(7) Alglave et al., *CAV*, 2010.
(8) Batty et al., *POPL*, 2011.
(9) Alglave et al., *ASPLOS*, 2015.

Compiling atomics

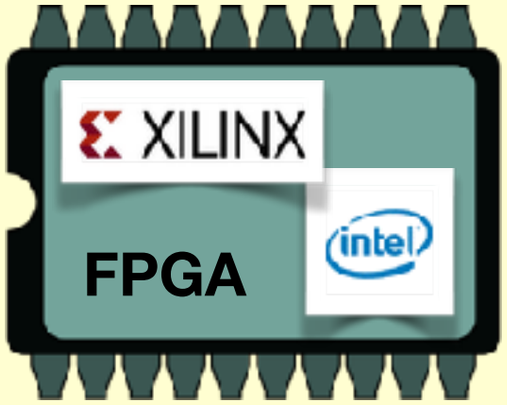


Compiling atomics




```
r = atomic_load(&y,
  memory_order_acquire);
```

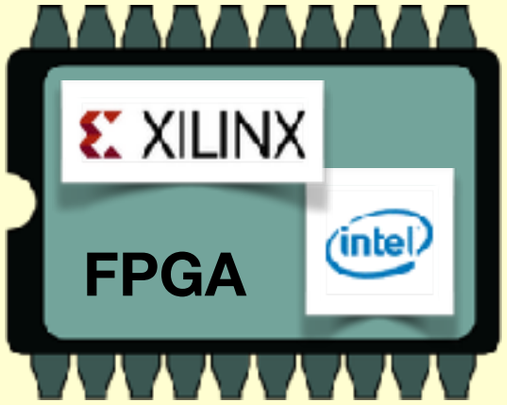
Compiling atomics




```
r = atomic_load(&y,
  memory_order_acquire);
```

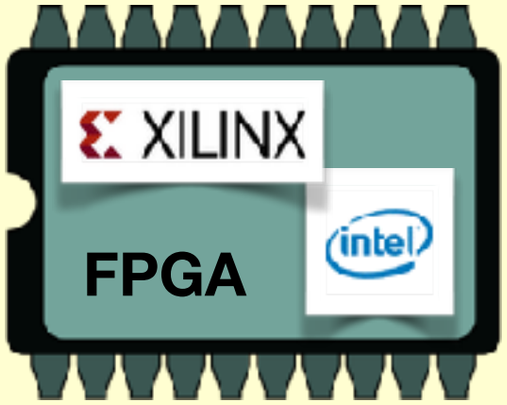
not supported

Compiling atomics




```
r = atomic_load(&y,
  memory_order_acquire);
```

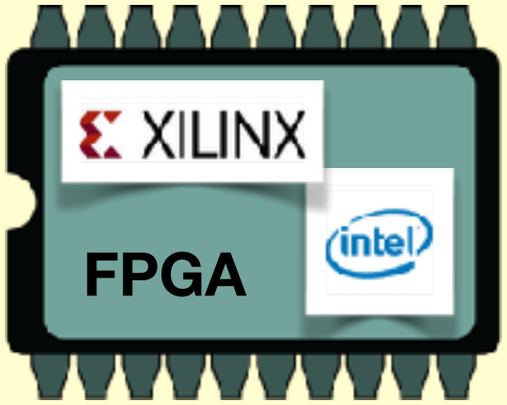
Compiling atomics




```
r = atomic_load(&y,
  memory_order_acquire);
```

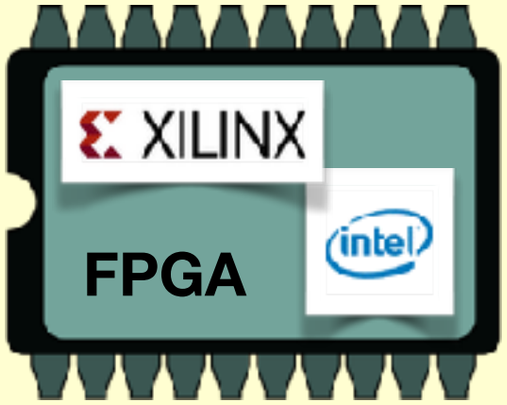
```
lock();
r = y;
unlock();
```

Compiling atomics



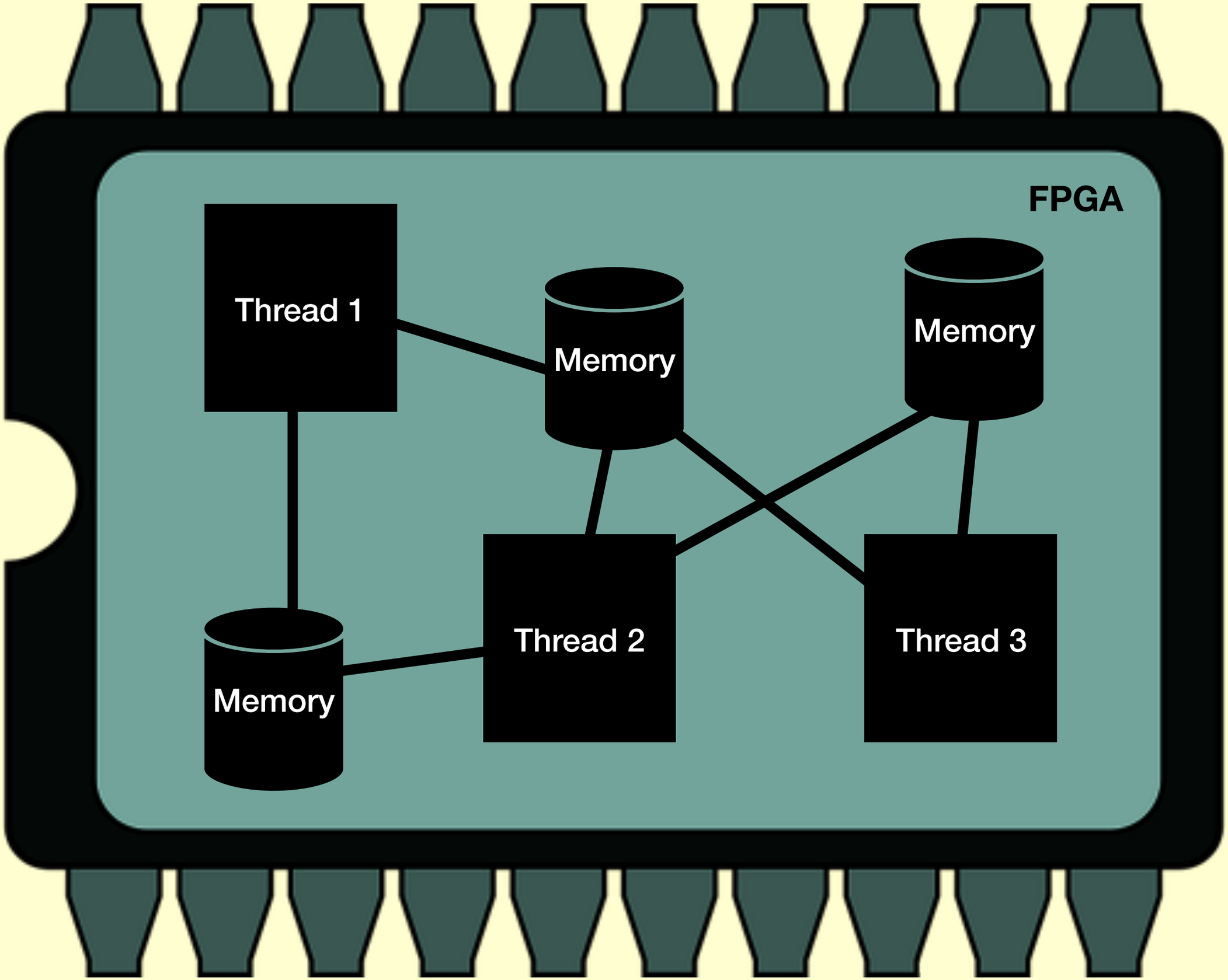

```
r = atomic_load(&y,
  memory_order_acquire);
```

Compiling atomics




```
r = atomic_load(&y,
  memory_order_acquire);
```

```
r = y;
```


FPGA
Thread 1
Memory
Memory
Memory
Thread 2
Thread 3

Atomic operations

- Atomics must appear to execute **instantaneously** to other threads
- Atomics provide a variety of **ordering** guarantees

Atomic operations

- Atomics must appear to execute **instantaneously** to other threads ✓
- Atomics provide a variety of **ordering** guarantees

Atomic operations

- Atomics must appear to execute **instantaneously** to other threads ✓
- Atomics provide a variety of **ordering** guarantees ?

```
r1 = atomic_load(&x,
  memory_order_relaxed);
r2 = atomic_load(&x,
  memory_order_relaxed);
```

```
atomic_store(&x, 1,
  memory_order_relaxed);
```

```
r1 = atomic_load(&x,
  memory_order_relaxed);
r2 = atomic_load(&x,
  memory_order_relaxed);
```

```
atomic_store(&x, 1,
  memory_order_relaxed);
```

```
r1 = x;
r2 = x;
```

```
x = 1;
```

```
r1 = atomic_load(&x,
  memory_order_relaxed);
r2 = atomic_load(&x,
  memory_order_relaxed);
```

```
atomic_store(&x, 1,
  memory_order_relaxed);
```

```
r1 = x;
r2 = x;
```

```
x = 1;
```

	1	2	3	4
`r1 = x;`	load x			
`r2 = x;`			load x	

```
r1 = atomic_load(&x,
  memory_order_relaxed);
r2 = atomic_load(&x,
  memory_order_relaxed);
```

```
atomic_store(&x, 1,
  memory_order_relaxed);
```

```
r1 = x;
r2 = x;
```

```
x = 1;
```

	1	2	3	4
`r1 = x;`	load x			
`r2 = x;`			load x	

	1
`x = 1;`	store x

```
r1 = atomic_load(&x,
  memory_order_relaxed);
r2 = atomic_load(&x,
  memory_order_relaxed);
```

```
atomic_store(&x, 1,
  memory_order_relaxed);
```

```
r1 = x;
r2 = x;
```

```
x = 1;
```

	1	2
`r1 = x;`	load x	
`r2 = x;`	load x	

	1
`x = 1;`	store x

```
r1 = atomic_load(&x,
  memory_order_relaxed);
r2 = atomic_load(&x,
  memory_order_relaxed);
```

```
atomic_store(&x, 1,
  memory_order_relaxed);
```

```
r1 = x;
r2 = x;
```

```
x = 1;
```

	1
`x = 1;`	store x

```
r1 = atomic_load(&x,
  memory_order_relaxed);
r2 = atomic_load(&x,
  memory_order_relaxed);
```

```
atomic_store(&x, 1,
  memory_order_relaxed);
```

```
r1 = x;
r2 = x/a;
```

```
x = 1;
```

	1
`x = 1;`	store x

```
r1 = atomic_load(&x,
  memory_order_relaxed);
r2 = atomic_load(&x,
  memory_order_relaxed);
```

```
atomic_store(&x, 1,
  memory_order_relaxed);
```

```
r0 = y+y+y+y+y+y;
r1 = x;
r2 = x/a;
```

```
x = 1;
```

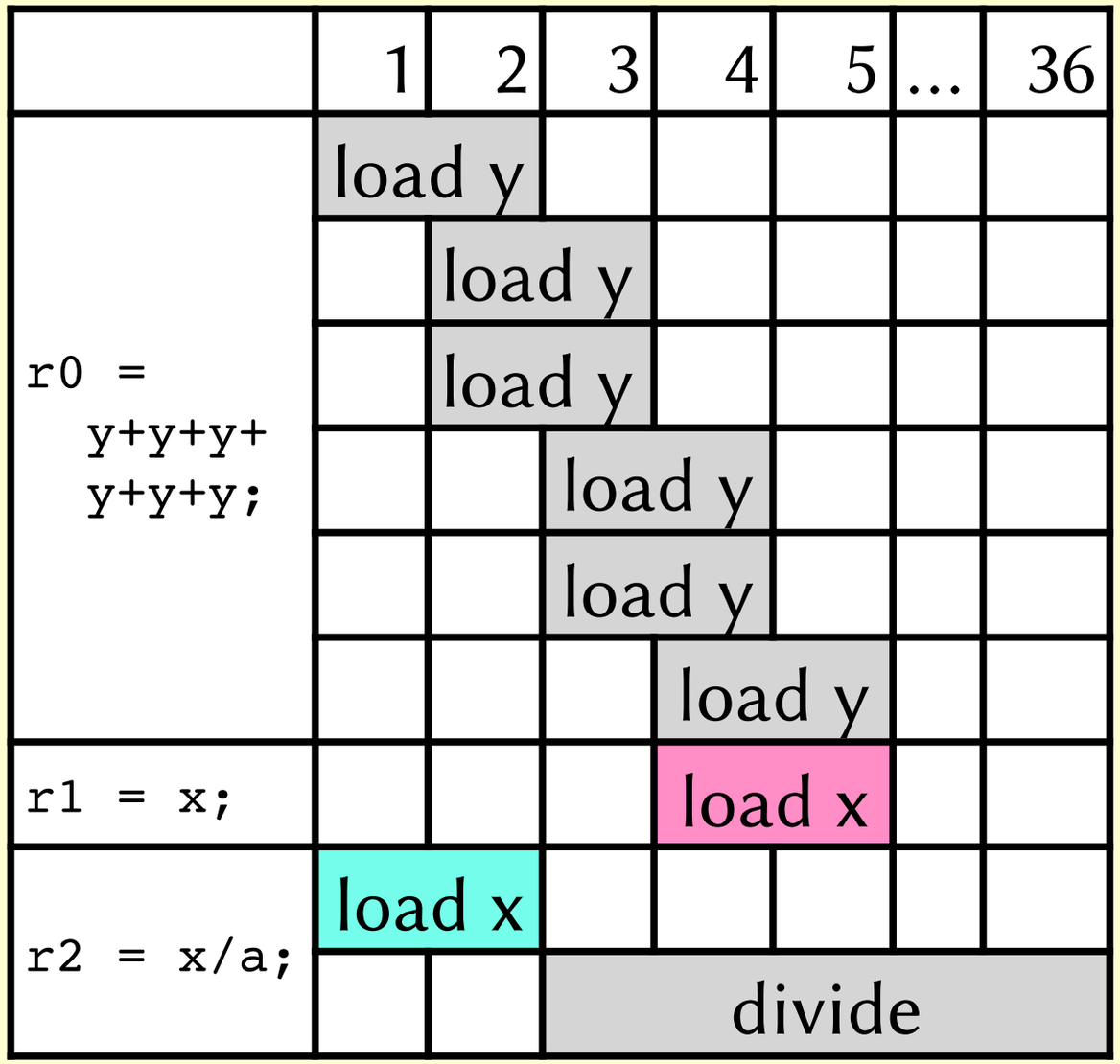
	1
x = 1;	store x

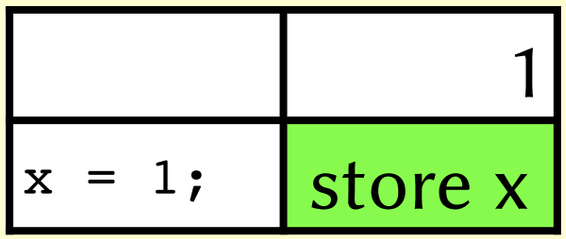
```
r1 = atomic_load(&x,
  memory_order_relaxed);
r2 = atomic_load(&x,
  memory_order_relaxed);
```

```
atomic_store(&x, 1,
  memory_order_relaxed);
```

```
r0 = y+y+y+y+y+y;
r1 = x;
r2 = x/a;
```

```
x = 1;
```





Constraints on scheduling

Constraints on scheduling

- Two atomic accesses to the same location cannot be reordered.

Constraints on scheduling

- Two atomic accesses to the same location cannot be reordered.
- An atomic acquire load cannot be reordered with accesses that come later in program order

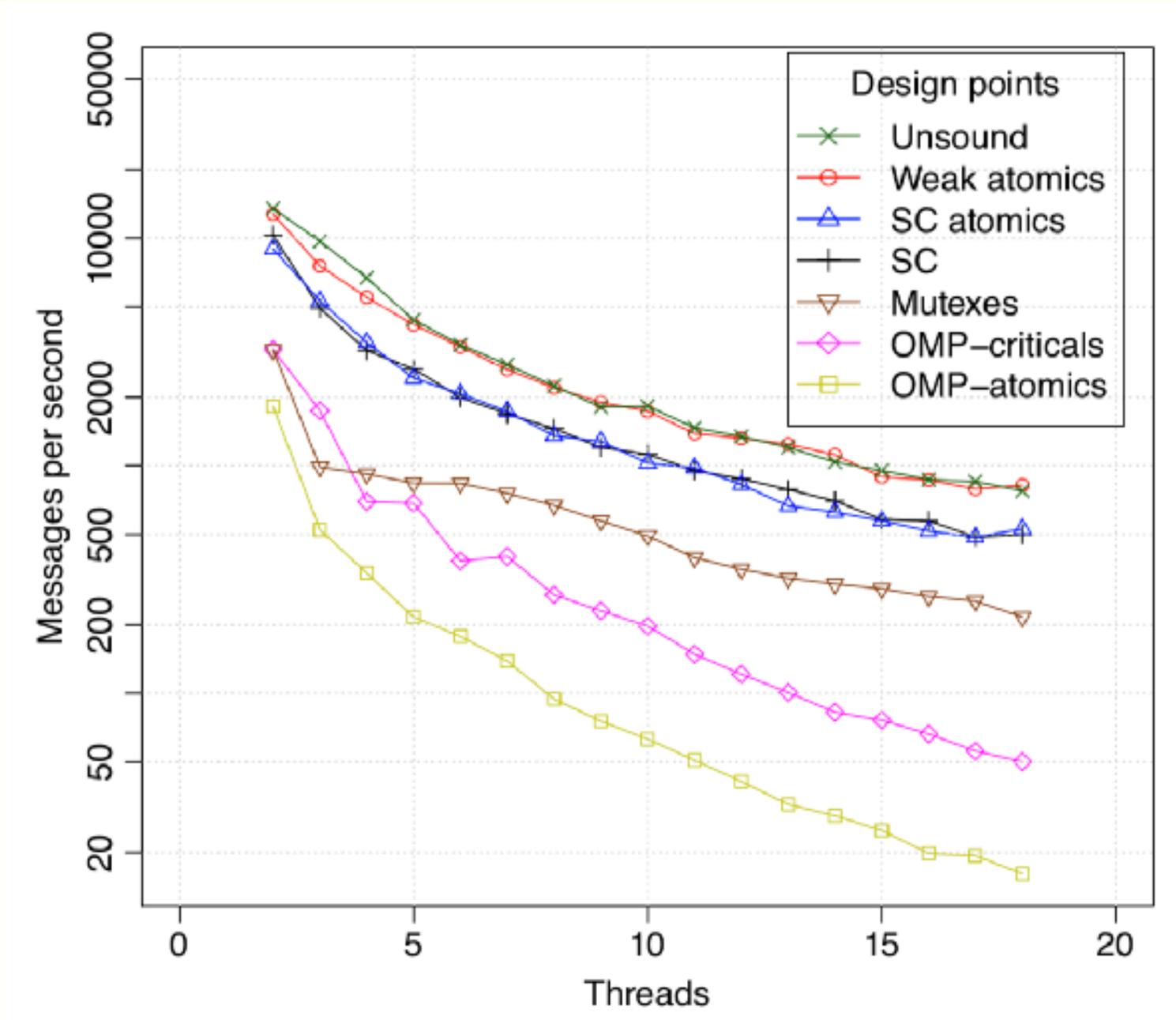
Constraints on scheduling

- Two atomic accesses to the same location cannot be reordered.
- An atomic acquire load cannot be reordered with accesses that come later in program order
- An atomic release store cannot be reordered with accesses that come earlier in program order

Constraints on scheduling

- Two atomic accesses to the same location cannot be reordered.
- An atomic acquire load cannot be reordered with accesses that come later in program order
- An atomic release store cannot be reordered with accesses that come earlier in program order
- An atomic SC access cannot be reordered with any other access.

Results



(10) Ramanathan et al., "Hardware Synthesis of Weakly Consistent C Concurrency", *FPGA*, 2017

Checking correctness

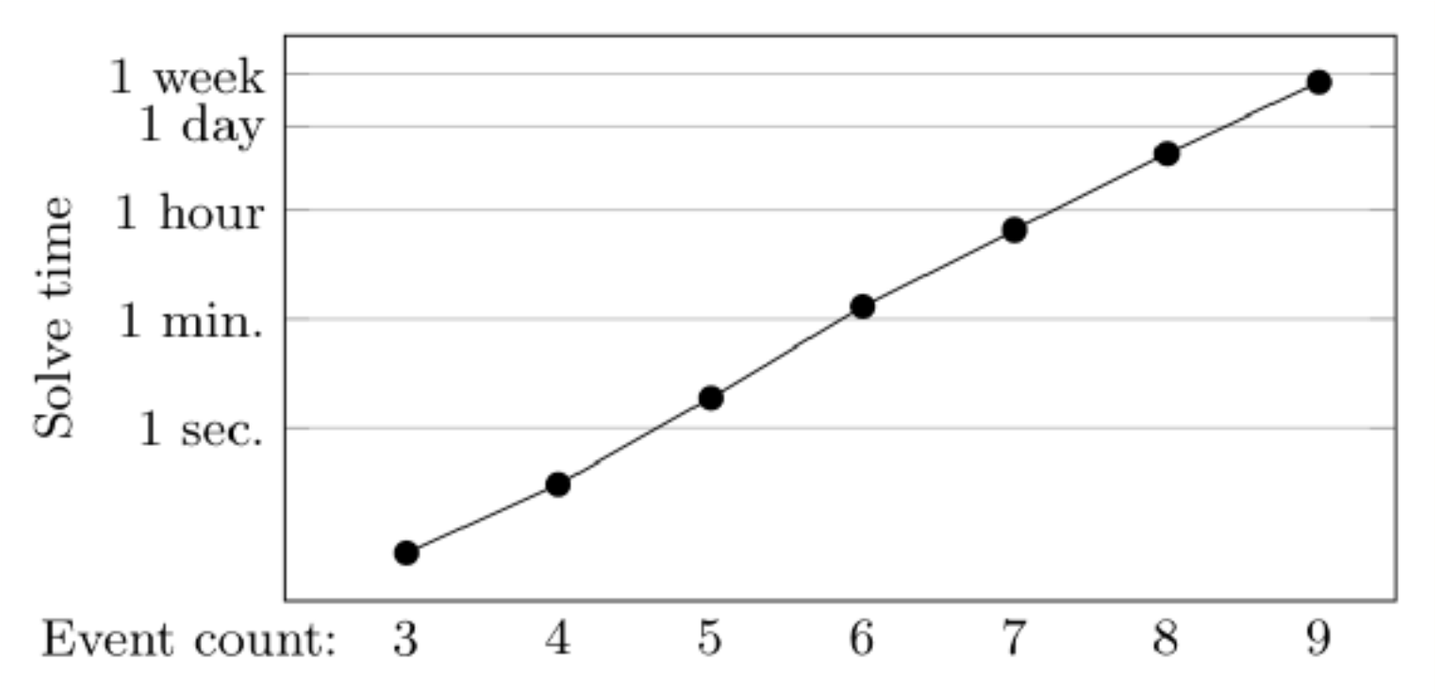
Checking correctness

- Ask Memalloy[11] for an execution that is **forbidden** according to the C++ standard but is **allowed** by our scheduling constraints.

(11) Wickerson et al., "Automatically Comparing Memory Consistency Models", *POPL*, 2017

Checking correctness

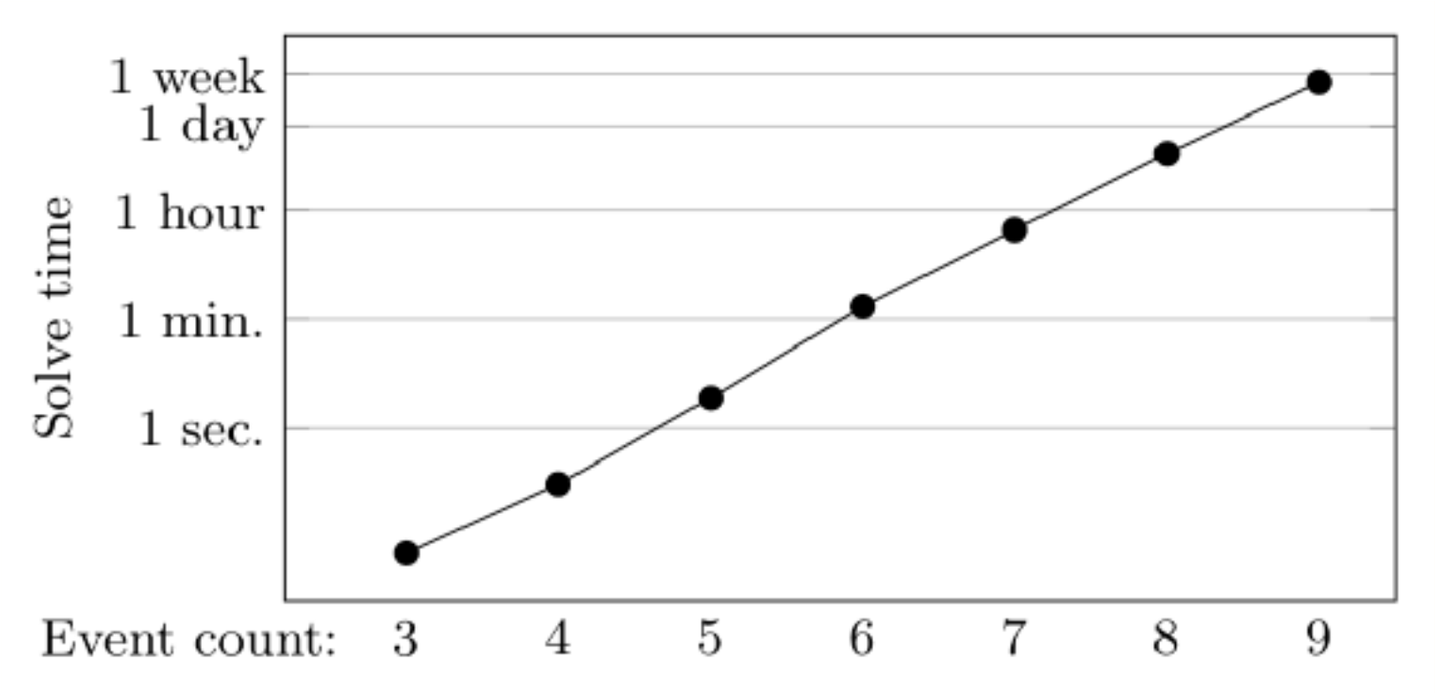
- Ask Memalloy[11] for an execution that is **forbidden** according to the C++ standard but is **allowed** by our scheduling constraints.



(11) Wickerson et al., "Automatically Comparing Memory Consistency Models", *POPL*, 2017

Checking correctness

- Ask Memalloy[11] for an execution that is **forbidden** according to the C++ standard but is **allowed** by our scheduling constraints.



- Memalloy uses the Alloy model checker, which in turn uses a SAT-solving backend.

(11) Wickerson et al., "Automatically Comparing Memory Consistency Models", *POPL*, 2017

Can we do better?

```
r1 = atomic_load(&x,
  memory_order_relaxed);
r2 = atomic_load(&x,
  memory_order_relaxed);
```

```
atomic_store(&x, 1,
  memory_order_relaxed);
```

	1	2
`r1 = x;`	load x	
`r2 = x;`	load x	

	1
`x = 1;`	store x

```
r1 = atomic_load(&x,
  memory_order_relaxed);
r2 = atomic_load(&x,
  memory_order_relaxed);
```

	1	2
`r1 = x;`	load x	
`r2 = x;`	load x	

Sync-aware scheduling

```
x = 1;

atomic_store(&xr, 1,
  memory_order_release);

y = 1;

atomic_store(&yr, 1,
  memory_order_release);
```

```
r = atomic_load(&xr,
  memory_order_acquire);

if (r==1) { print(x); }
```

```
s = atomic_load(&yr,
  memory_order_acquire);

if (s==1) { print(y); }
```

Sync-aware scheduling

```
x = 1;

atomic_store(&xr, 1,
  memory_order_release);

y = 1;

atomic_store(&yr, 1,
  memory_order_release);
```

```
r = atomic_load(&xr,
  memory_order_acquire);

if (r==1) { print(x); }
```

```
s = atomic_load(&yr,
  memory_order_acquire);

if (s==1) { print(y); }
```

Sync-aware scheduling

```
x = 1;

atomic_store(&xr, 1,
  memory_order_release);

y = 1;

atomic_store(&yr, 1,
  memory_order_release);
```

```
r = atomic_load(&xr,
  memory_order_acquire);

if (r==1) { print(x); }
```

```
s = atomic_load(&yr,
  memory_order_acquire);

if (s==1) { print(y); }
```

Sync-aware scheduling

```
x = 1;

atomic_store(&xr, 1,
  memory_order_release);

y = 1;

atomic_store(&yr, 1,
  memory_order_release);
```

```
r = atomic_load(&xr,
  memory_order_acquire);

if (r==1) { print(x); }
```

```
s = atomic_load(&yr,
  memory_order_acquire);

if (s==1) { print(y); }
```

Sync-aware scheduling

```
x = 1;

atomic_store(&xr, 1,
  memory_order_release);

y = 1;

atomic_store(&yr, 1,
  memory_order_release);
```

```
r = atomic_load(&xr,
  memory_order_acquire);

if (r==1) { print(x); }
```

```
s = atomic_load(&yr,
  memory_order_acquire);

if (s==1) { print(y); }
```

Sync-aware scheduling

```
x = 1;

atomic_store(&xr, 1,
  memory_order_release);

y = 1;

atomic_store(&yr, 1,
  memory_order_release);
```

```
r = atomic_load(&xr,
  memory_order_acquire);

if (r==1) { print(x); }
```

```
s = atomic_load(&yr,
  memory_order_acquire);

if (s==1) { print(y); }
```

Sync-aware scheduling

```
x = 1;

atomic_store(&xr, 1,
  memory_order_release);

y = 1;

atomic_store(&yr, 1,
  memory_order_release);
```

```
r = atomic_load(&xr,
  memory_order_acquire);

if (r==1) { print(x); }
```

```
s = atomic_load(&yr,
  memory_order_acquire);

if (s==1) { print(y); }
```

Longer paths too

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
```

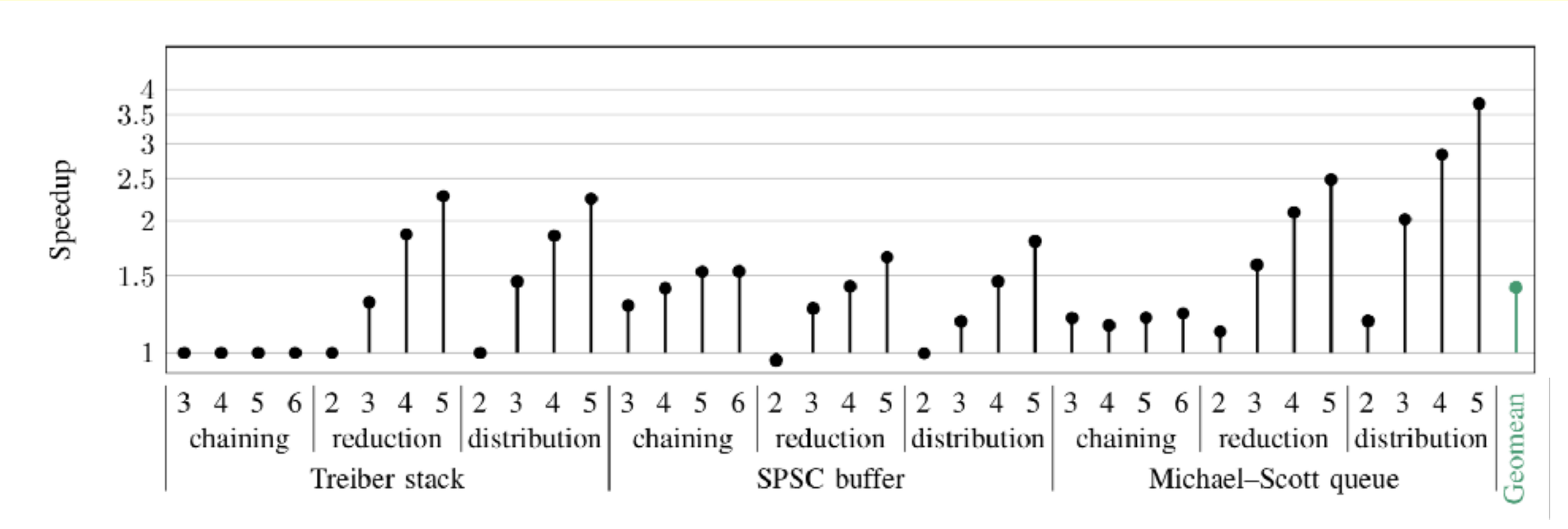
```
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Results



(12) Ramanathan et al., "Concurrency-Aware Thread Scheduling for High-Level Synthesis", *FCCM*, 2018

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

- Our solution: enumerate only the "primary" paths.

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

Poorly scaling analysis

```
x = 1;

atomic_store(&y, 1,
  memory_order_release);
atomic_store(&y, 1,
  memory_order_release);
```

```
r = atomic_load(&y,
  memory_order_acquire);
r = atomic_load(&y,
  memory_order_acquire);

if (r==1) {
 atomic_store(&z, 1,
  memory_order_release);
 atomic_store(&z, 1,
  memory_order_release);
}
```

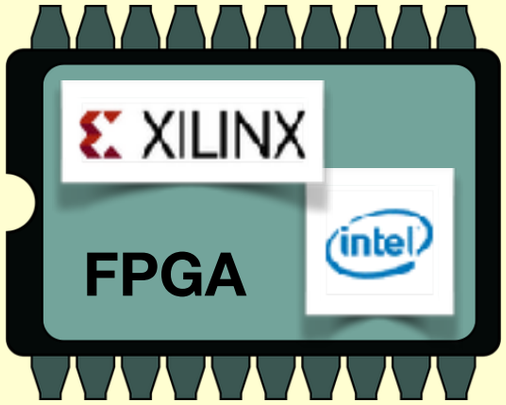
```
s = atomic_load(&z,
  memory_order_acquire);
s = atomic_load(&z,
  memory_order_acquire);

if (s==1) { print(x); }
```

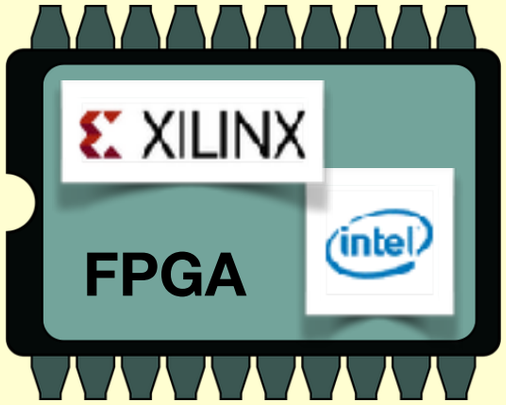
Checking correctness

- As before, we use Memalloy to check that our constraints are strong enough to guarantee C++ semantics.

Where next?

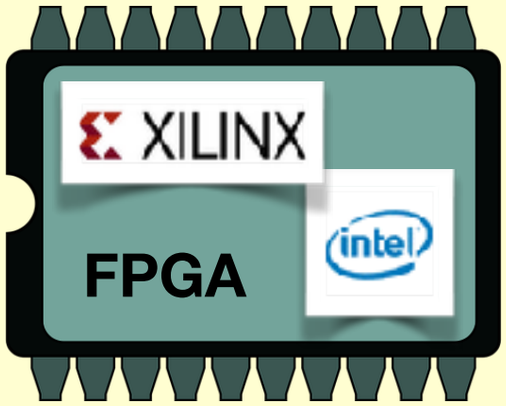


Where next?



- **Heavyweight:** a fully verified hardware compiler (e.g. a Verilog backend for CompCert).

Where next?



- **Heavyweight:** a fully verified hardware compiler (e.g. a Verilog backend for CompCert).

- **Lightweight:** automatically generate and verify SystemVerilog assertions, *à la* RTLCheck.(13)

(13) Manerkar et al., "RTLCheck: Verifying the Memory Consistency of RTL Designs", *MICRO*, 2017.

Towards Verified Hardware Compilation

John Wickerson
Imperial College London

FMATS Workshop, Microsoft Research Cambridge, 24 Sep 2018