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.

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.
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.
Hardware Compilation of Concurrency

- C++ Programming Language
- OpenCL
- CPU
- GPU
- FPGA
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

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

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

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

```c
x = 1;
atomic_store(&y, 1, memory_order_release);

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

atomic_store(&x, 1, memory_order_relaxed);

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

```c
atomic_store(&x, 1, memory_order_relaxed);
```
Weak-memory concurrency is tricky!
Weak-memory concurrency is tricky!

- x86 proved tricky to formalise correctly.\(^{(5,6)}\)

(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.\(^{(7)}\)

(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.\(^{(7)}\)

- C++ specification did not guarantee its own key property.\(^{(8)}\)

---

(6) Owens et al., *TPHOLs*, 2009.
(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.\(^{(9)}\)

---

(6) Owens et al., *TPHOLs*, 2009.
(8) Batty et al., *POPL*, 2011.
(9) Alglave et al., *ASPLOS*, 2015.
Compiling atomics
Compiling atomics

```c
r = atomic_load(&y, memory_order_acquire);
```
Compiling atomics

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

**not supported**
Compiling atomics

\[ r = \text{atomic\_load}(&y, \text{memory\_order\_acquire}); \]
Compiling atomics

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

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

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

\[
r = \text{atomic\_load}(\&y, \\
\text{memory\_order\_acquire});
\]

\[
r = y;
\]
FPGA

Thread 1

Thread 2

Thread 3

Memory

Memory

Memory
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 \textit{instantaneously} to other threads

- Atomics provide a variety of \textit{ordering} guarantees
Atomic operations

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

- Atomics provide a variety of *ordering* guarantees
\begin{Verbatim}
r1 = atomic_load(&x, memory_order_relaxed);
r2 = atomic_load(&x, memory_order_relaxed);
\end{Verbatim}

\begin{Verbatim}
atomic_store(&x, 1, memory_order_relaxed);
\end{Verbatim}
\begin{verbatim}
  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;
\end{verbatim}
\[ r1 = \text{atomic\_load}(&x, \text{memory\_order\_relaxed}); \]
\[ r2 = \text{atomic\_load}(&x, \text{memory\_order\_relaxed}); \]
\[ \text{atomic\_store}(&x, 1, \text{memory\_order\_relaxed}); \]
\[ r1 = \text{atomic\_load}(&x, \text{memory\_order\_relaxed}); \]
\[ r2 = \text{atomic\_load}(&x, \text{memory\_order\_relaxed}); \]
\[ x = 1; \]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = x;</td>
<td></td>
<td>load x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2 = x;</td>
<td></td>
<td></td>
<td>load x</td>
<td></td>
</tr>
</tbody>
</table>
\begin{align*}
    r1 &= \texttt{atomic\_load}(&x, \\
    &\quad \text{memory\_order\_relaxed}); \\
    r2 &= \texttt{atomic\_load}(&x, \\
    &\quad \text{memory\_order\_relaxed}); \\
    \texttt{atomic\_store}(&x, 1, \\
    &\quad \text{memory\_order\_relaxed}); \\
\end{align*}

\begin{align*}
    r1 &= x; \\
    r2 &= x; \\
    x &= 1; \\
\end{align*}
r1 = \texttt{atomic\_load}(&x, memory\_order\_relaxed);
r2 = \texttt{atomic\_load}(&x, memory\_order\_relaxed);\hline
\texttt{atomic\_store}(&x, 1, memory\_order\_relaxed);

r1 = x;
r2 = x;\hline
x = 1;
\begin{align*}
\texttt{r1} &= \texttt{atomic\_load}(&x, \\
                  &\texttt{memory\_order\_relaxed}); \\
\texttt{r2} &= \texttt{atomic\_load}(&x, \\
                  &\texttt{memory\_order\_relaxed}); \\
\texttt{atomic\_store}(&\texttt{x, 1}, \\
                   &\texttt{memory\_order\_relaxed}); \\
\texttt{r1} &= \texttt{atomic\_load}(&\texttt{x}, \\
                   &\texttt{memory\_order\_relaxed}); \\
\texttt{r2} &= \texttt{atomic\_load}(&\texttt{x}, \\
                   &\texttt{memory\_order\_relaxed}); \\
\texttt{x} &= 1;
\end{align*}

\begin{tabular}{|c|}
\hline
\texttt{x} = 1; \\
\hline
\end{tabular}

\textcolor{green!50!black}{\texttt{store} \texttt{x}}
\begin{align*}
    r1 &= \text{atomic\_load}(&x, \text{memory\_order\_relaxed}); \\
    r2 &= \text{atomic\_load}(&x, \text{memory\_order\_relaxed}); \\
    r1 &= x; \\
    r2 &= x/a;
\end{align*}

\begin{align*}
    \text{atomic\_store}(&x, 1, \text{memory\_order\_relaxed}); \\
    x &= 1;
\end{align*}

\begin{tabular}{|c|c|}
    \hline
    x = 1; & store x \\
    \hline
\end{tabular}
```c
x = 1;
store x
```

```c
1
```

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

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

```c
atomic_store(&x, 1, memory_order_relaxed);
```
\[ r_0 = y + y + y + y + y + y; \]
\[ r_1 = x; \]
\[ r_2 = x / a; \]

\[
\begin{align*}
\text{atomic_load}(&x, \text{memory_order_relaxed}); \\
\text{atomic_store}(&x, 1, \text{memory_order_relaxed}); \\
\end{align*}
\]

\[
\begin{align*}
x = 1; \\
\end{align*}
\]
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\(^{(1)}\) 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.

\(^{(1)}\) Wickerson et al., "Automatically Comparing Memory Consistency Models", *POPL*, 2017
Can we do better?
\begin{verbatim}
\begin{align*}
  r1 & = \text{atomic\_load}(&x, \text{memory\_order\_relaxed}); \\
  r2 & = \text{atomic\_load}(&x, \text{memory\_order\_relaxed}); \\
  \text{atomic\_store}(&x, 1, \text{memory\_order\_relaxed}); \\
\end{align*}
\end{verbatim}
\begin{verbatim}
    r1 = atomic_load(&x, memory_order_relaxed);
    r2 = atomic_load(&x, memory_order_relaxed);
\end{verbatim}
Sync-aware scheduling

\[
\begin{align*}
\text{x} &= 1; \\
\text{atomic\_store}(&\text{xr}, 1, \\
\text{memory\_order\_release}); \\
\text{y} &= 1; \\
\text{atomic\_store}(&\text{yr}, 1, \\
\text{memory\_order\_release}); \\
\text{r} &= \text{atomic\_load}(&\text{xr}, \\
\text{memory\_order\_acquire}); \\
\text{if} (r==1) \{ \text{print}(x); \} \\
\text{s} &= \text{atomic\_load}(&\text{yr}, \\
\text{memory\_order\_acquire}); \\
\text{if} (s==1) \{ \text{print}(y); \}
\end{align*}
\]
Sync-aware scheduling

```c
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

\[
x = 1;
\]
\[
\text{atomic}_{\text{store}}(\&xr, 1, \text{memory}_{\text{order}}_{\text{release}});
\]
\[
y = 1;
\]
\[
\text{atomic}_{\text{store}}(\&yr, 1, \text{memory}_{\text{order}}_{\text{release}});
\]
\[
r = \text{atomic}_{\text{load}}(\&xr, \text{memory}_{\text{order}}_{\text{acquire}});
\]
\[
\text{if } (r==1) \{ \text{print}(x); \}
\]
\[
s = \text{atomic}_{\text{load}}(\&yr, \text{memory}_{\text{order}}_{\text{acquire}});
\]
\[
\text{if } (s==1) \{ \text{print}(y); \}
Sync-aware scheduling

\[
x = 1;
\text{atomic\_store}(&xr, 1, memory\_order\_release);
\]

\[
y = 1;
\text{atomic\_store}(&yr, 1, memory\_order\_release);
\]

\[
r = \text{atomic\_load}(&xr, memory\_order\_acquire);
\text{if (r==1) \{ print(x); \}}
\]

\[
s = \text{atomic\_load}(&yr, memory\_order\_acquire);
\text{if (s==1) \{ print(y); \}}
\]
Sync-aware scheduling

\[ x = 1; \]
\[ \text{atomic\_store}(&xr, 1, \text{memory\_order\_release}); \]
\[ y = 1; \]
\[ \text{atomic\_store}(&yr, 1, \text{memory\_order\_release}); \]
\[ r = \text{atomic\_load}(&xr, \text{memory\_order\_acquire}); \]
\[ \text{if} (r==1) \{ \text{print}(x); \} \]
\[ s = \text{atomic\_load}(&yr, \text{memory\_order\_acquire}); \]
\[ \text{if} (s==1) \{ \text{print}(y); \} \]
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); }

s = atomic_load(&yr, memory_order_acquire);
if (s==1) { print(y); }
Sync-aware scheduling

```c
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); }
```
Towards Verified Hardware Compilation

Longer paths too

```c
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

```c
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); }
```
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

```c
x = 1;
atomic_store(&y, 1, memory_order_release);
atomic_store(&y, 1, memory_order_release);
if (r==1) {
    atomic_store(&z, 1, memory_order_release);
    atomic_store(&z, 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

```c
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

```c
x = 1;
atomic_store(&y, 1, memory_order_release);
atomic_store(&y, 1, memory_order_release);
if (r==1) {
    atomic_store(&z, 1, memory_order_release);
    atomic_store(&z, 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

```c
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

```c
x = 1;
atomic_store(&y, 1, memory_order_release);
atomic_store(&y, 1, memory_order_release);
```

```c
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);
}
```

```c
s = atomic_load(&z, memory_order_acquire);
s = atomic_load(&z, memory_order_acquire);
if (s==1) { print(x); }
```
Poorly scaling analysis

```c
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

```c
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

```c
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); }
```
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

```c
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); }```

```c
```
Poorly scaling analysis

```c
x = 1;
atomic_store(&y, 1, memory_order_release);
atomic_store(&y, 1, memory_order_release);

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

```c
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

```c
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

```c
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); }
```
x = 1;
atomic_store(&y, 1, memory_order_release);
atomic_store(&y, 1, memory_order_release);
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); }

r = atomic_load(&y, memory_order_acquire);
r = atomic_load(&y, memory_order_acquire);
Poorly scaling analysis

- Our solution: enumerate only the "primary" paths.
Poorly scaling analysis

```c
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

```c
x = 1;
atomic_store(&y, 1, memory_order_release);
atomic_store(&y, 1, memory_order_release);
```

```c
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

```c
x = 1;
atomic_store(&y, 1, memory_order_release);
atomic_store(&y, 1, memory_order_release);
```

```c
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);
}
```

```c
s = atomic_load(&z, memory_order_acquire);
s = atomic_load(&z, memory_order_acquire);
if (s==1) { print(x); }
```
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); }
Towards Verified Hardware Compilation

Poorly scaling analysis

```
x = 1;
atomic_store(&y, 1, memory_order_release);
atomic_store(&y, 1, memory_order_release);
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); }```

```
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);
} ```
Poorly scaling analysis

John Wickerson
Towards Verified Hardware Compilation

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

```c
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

```c
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.\(^{(13)}\)

Towards Verified Hardware Compilation

John Wickerson
Imperial College London

FMATS Workshop, Microsoft Research Cambridge, 24 Sep 2018