Hardware Synthesis of Weakly Consistent C Concurrency

a talk based on joint work with

Nadesh Ramanathan
Imperial College London

Shane T. Fleming
Imperial College London

George A. Constantinides
Imperial College London

John Wickerson
Imperial College London

S-REPLS @ Oxford

12 Jan 2017
Starting point

Concurrent languages with built-in support for strong/weak atomics

CPUs
- Intel
- ARM

GPUs
- AMD
- NVIDIA

FPGAs
- Xilinx
- ALTERA

OpenCL
FPGAs

- Reconfigurable hardware
Intel Completes Acquisition of Altera

$16.7 billion deal underscores Intel CEO’s plan to expand chip maker’s business

By DON CLARK
Updated Dec. 28, 2015 9:27 p.m. ET

Intel Corp. on Monday completed its biggest-ever acquisition, part of Chief Executive Brian Krzanich’s plan to use new tactics to expand the chip maker’s business.

The $16.7 billion purchase of Altera Corp. makes Intel, known for microprocessors used in computers, the second-largest maker of programmable chips, which are programmed after they leave the factory. Altera’s
LegUp

• Open-source hardware synthesis tool developed at the University of Toronto since 2009

• Supports pthreads, and OpenMP (using locks)
Synthesis example

- Can we implement atomic stores/loads using just ordinary stores/loads?

```plaintext
int x=0; int y=0;

T1() {
  1.1 x=1;
  1.2 y=1;
}

T2() {
  2.1 if(y==1)  
  2.2 r0=x;
}

assert(r0 \neq 0)
```
Synthesis example

- Can we implement atomic stores/loads using just ordinary stores/loads?

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

T1(int a) {
    1.1 x=a/3;
    1.2 y=1;
}

T2() {
    2.1 if(y==1)
    2.2 r0=x;
}

assert(r0 != 0)
```
```
int x=0; int y=0;

T1(int a) {
1.1  x=a/3;
1.2  y=1;
}

T2() {
2.1  if(y==1)
2.2  r0=x;
}

assert(r0 ≠ 0)
```

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
<th>35</th>
<th>36</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>ld a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td></td>
<td></td>
<td>divide</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td>st x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>st y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td></td>
<td></td>
<td>ld y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td></td>
<td></td>
<td>ld x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td></td>
<td></td>
<td>slt y==1?</td>
<td>x:null</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementing atomics

- **Unsound**: only respects RW/WR/WW dependencies

```c
r0=w;
r1=x;
r2=y.ld(ACQ);
r3=z;
```
### Implementing atomics

**Unsound:** only respects RW/WR/WW dependencies

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0=w;</td>
<td>ld&lt;sub&gt;na&lt;/sub&gt; w</td>
<td></td>
</tr>
<tr>
<td>r1=x;</td>
<td>ld&lt;sub&gt;na&lt;/sub&gt; x</td>
<td></td>
</tr>
<tr>
<td>r2=y.ld(ACQ);</td>
<td>ld&lt;sub&gt;ACQ&lt;/sub&gt; y</td>
<td></td>
</tr>
<tr>
<td>r3=z;</td>
<td>ld&lt;sub&gt;na&lt;/sub&gt; z</td>
<td></td>
</tr>
</tbody>
</table>
Implementing atomics

- **SC:** all memory accesses strictly ordered

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0=w;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld_{na} w</td>
</tr>
<tr>
<td>r1=x;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld_{na} x</td>
<td></td>
</tr>
<tr>
<td>r2=y. ld(ACQ);</td>
<td></td>
<td></td>
<td></td>
<td>ld_{ACQ} y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3=z;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld_{na} z</td>
</tr>
</tbody>
</table>

```c
int x=0; atomic_int y=0;
r2=y.ld(ACQ);
r3=z;
```

We propose three different methods to implement atomics. We compare-and-swap (CAS) results are not consistent, but our full implementation also includes fences [1]. We do not consider atomic read-modify-write instructions (such as pHMEM) in our implementation. We build on the LegUp framework, as it only synchronises using atomics rather than locks. We then evaluate the consistency of the memory operations using an MCM generated from the LLVM IR. We implement our methods on the LegUp framework and evaluate the impact of our approach on HLS tools that run on the LegUp framework. The proposed approach is generally applicable to HLS tools.
Implementing atomics

- **SC-atomics:** all atomics are strictly ordered

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0=w;</td>
<td></td>
<td>ld_{na} w</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r1=x;</td>
<td></td>
<td>ld_{na} x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2=y.ld(ACQ);</td>
<td></td>
<td></td>
<td></td>
<td>ld_{ACQ} y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3=z;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld_{na} z</td>
</tr>
</tbody>
</table>
Implementing atomics

- **Weak-atomics**: acquires cannot move down; releases cannot move up; SCs cannot move at all.

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0=w;</td>
<td></td>
<td>$ld_{na}~w$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r1=x;</td>
<td></td>
<td>$ld_{na}~x$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2=y. $ld(ACQ)$;</td>
<td></td>
<td>$ld_{ACQ}~y$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3=z;</td>
<td></td>
<td></td>
<td></td>
<td>$ld_{na}~z$</td>
</tr>
</tbody>
</table>
Correctness

- We used the Alloy Analyzer to check that there is no execution (with $\leq 9$ memory events) that:
  
  - is allowed by our scheduling constraints, but
  
  - is inconsistent according to the C memory model.
Case study: circular FIFO

```c
atomic_int tail=0; head=0;
int arr[SIZE]; res[MSGS];

1.1 while(prod<MSGS) {
1.2   chead = head.ld(ACQ);
1.3   ctail = tail.ld(RLX);
1.4   ntail = (ctail+1)%SIZE;
1.5   if(ntail != chead){
1.6     arr[ctail] = prod
1.7     tail.st(ntail,REL);
1.8     prod++;
1.9   }
1.10 }

2.1 while(cons<MSGS) {
2.2   ctail = tail.ld(ACQ);
2.3   chead = head.ld(RLX);
2.4   nhead = (chead+1)%SIZE;
2.5   if(ctail != chead){
2.6     res[cons] = arr[chead];
2.7     head.st(nhead,REL);
2.8     cons++;
2.9   }
2.10 }
```

Figure 5 shows the C-like code of a producer (on the left) and consumer (on the right) communicating via a circular buffer. The diagrammatic example: a lock-free single-producer-single-consumer (SPSC) circular buffer, similar to this circular buffer, is implemented with the MCMs discussed in §5.1. The first version, could lead to incorrect results. We explain in detail why each memory access does not race with the consumer only removes tasks, as reflected by the store for release, or the producer writes to stores to acquire, at 64 and the number of messages transmitted (update, as depicted in Figure 6. We fix the buffer size (by the producer in line 1.6) do not race with the actual code we verified online [1].

C++ library and the Linux kernel [5].

In our evaluation, we investigate the performance of SC atomics and weak atomics on a real-world example: a lock-free single-producer-single-consumer (SPSC) circular buffer. The producer only adds tasks and the consumer only removes tasks, as reflected by the store for release, or the producer writes to stores to acquire, at 64 and the number of messages transmitted (update, as depicted in Figure 6. We fix the buffer size (by the producer in line 1.6) do not race with the actual code we verified online [1].
6. CONCLUSION

This work has investigated how to implement lock-free algorithms on FPGAs using HLS. Our case study suggests that careful reasoning about memory consistency, as opposed to relying on locks, allows us to recover most of the performance of unsound implementations, while guaranteeing correctness. Even our worst-case lock-free implementation (SC in Table 1) is on average 2.5x faster than our best-case lock-based implementation (Mutexes). We have also shown that weakly consistent atomics have a smaller performance overhead than sequentially consistent atomics.

We hope our work will stimulate further support in HLS tools for fine-grained synchronisation in multi-threaded C programs, and raise awareness of the possibility of synthesis.
6. CONCLUSION

This work has investigated how to implement lock-free algorithms on FPGAs using HLS. Our case study suggests that careful reasoning about memory consistency, as opposed to relying on locks, allows us to recover most of the performance of unsound implementations, while guaranteeing correctness. Even our worst-case lock-free implementation (SC in Table 1) is on average 2.5x faster than our best-case lock-based implementation (Mutexes). We have also shown that weakly consistent atomics have a smaller performance overhead than sequentially consistent atomics.

We hope our work will stimulate further support in HLS tools for fine-grained synchronisation in multi-threaded C programs, and raise awareness of the possibility of synthesis-
Conclusion

• First implementation of weak atomics in a hardware synthesis tool

• Implementing atomics using scheduling constraints seems more efficient than using locks

• **Limitations**: no support for RMW operations; small and artificial benchmarks; only on-chip memory

• **Next steps**: add support for loop pipelining
Hardware Synthesis of Weakly Consistent C Concurrency

a talk based on joint work with

Nadesh Ramanathan
Imperial College London

Shane T. Fleming
Imperial College London

George A. Constantinides
Imperial College London

S-REPLS @ Oxford 12 Jan 2017