Hardware Synthesis of Weakly Consistent C Concurrency

Nadesh Ramanathan
Imperial College London
n.ramanathan14@imperial.ac.uk

Shane T. Fleming
Imperial College London
shane.fleming06@imperial.ac.uk

John Wickerson
Imperial College London
j.wickerson@imperial.ac.uk

George A. Constantinides
Imperial College London
g.constantinides@imperial.ac.uk

ABSTRACT

Lock-free algorithms, in which threads synchronise not via coarse-grained mutual exclusion but via fine-grained atomic operations ('atomics'), have been shown empirically to be the fastest class of multi-threaded algorithms in the realm of conventional processors. This paper explores how these algorithms can be compiled from C to reconfigurable hardware via *high-level synthesis* (HLS).

We focus on the scheduling problem, in which software instructions are assigned to hardware clock cycles. We first show that typical HLS scheduling constraints are insufficient to implement atomics, because they permit some instruction reorderings that, though sound in a single-threaded context, demonstrably cause erroneous results when synthesising multi-threaded programs. We then show that correct behaviour can be restored by imposing additional *intra-thread* constraints among the memory operations. We implement our approach in the open-source LegUp HLS framework, and provide both *sequentially consistent* (SC) and *weakly consistent* ('weak') atomics. Weak atomics necessitate fewer constraints than SC atomics, but suffice for many concurrent algorithms. We confirm, via automatic model-checking, that we correctly implement the semantics defined by the 2011 revision of the C standard. A case study on a circular buffer suggests that circuits synthesised from programs that use atomics can be 2.5x faster than those that use locks, and that weak atomics can yield a further 1.5x speedup.

Keywords

atomic operations, C/C++, FPGAs, high-level synthesis, lock-free algorithms, memory consistency models, scheduling.

1. INTRODUCTION

Gramoli [13] demonstrates in his comprehensive empirical study that, when writing multi-threaded programs for conventional multi-processors, the most efficient way to synchronise threads is to use fine-grained *atomic operations* ('atomics') – as opposed to, for instance, coarse-grained mutual exclusion based on locks. In this paper, we explore how lock-free programs can be compiled from C to reconfigurable hardware via *high-level synthesis* (HLS), and the performance benefits of doing so.

We focus on the *scheduling* stage of synthesis, in which software instructions are assigned to hardware clock cycles. Typical HLS schedulers seek to maximise instruction-level parallelism by allowing independent instructions to be executed out-of-order or simultaneously. In particular, non-aliasing memory accesses, or those that exhibit only read-after-read (RAR) dependencies (e.g. `x=z; y=z`), can be reordered. These reorderings are invisible in a single-threaded context, but in a multi-threaded context, they can introduce unexpected behaviours. For instance, if another thread is simultaneously writing to `z`, then reordering may introduce the behaviour where `x` is assigned the latest value but `y` gets an old one.

The implication of this is not that HLS tools are *wrong*, because these optimisations can only introduce new behaviours when the code already exhibits a race condition, and races are deemed a programming error in C [17, §5.1.2.4]. Rather the implication is that if these memory accesses are upgraded to become atomic (and hence allowed to race), then existing scheduling constraints are insufficient.

One approach for implementing atomics correctly is to enclose each in its own critical region, and ensure that the surrounding `lock()` and `unlock()` calls cannot be reordered. We show that this approach, which is the only approach available in LegUp [7], is expensive and scales poorly. Instead, we frame the implementation of atomics as a scheduling problem: we treat atomic accesses as regular memory accesses but impose additional intra-thread dependencies when devising a schedule for each thread.

By default, C atomics enforce *sequential consistency* (SC), which means that all threads have a completely consistent view of shared memory, and memory accesses always occur in the order specified by the programmer [20]. Although simple for programmers to understand, SC is an expensive illusion for language implementations to maintain in the presence of optimisations by compilers (such as constant propagation) and by architectures (such as store buffering) that confound SC.

In fact, many concurrent algorithms do not need all threads to share a completely consistent view of shared memory, and hence can tolerate *weakly consistent* atomics, which do not provide this guarantee in general. These 'weak atomics' include the *acquire/release* and *relaxed* atomics provided by

FPGA '17, February 22-24, 2017, Monterey, CA, USA

DOI: http://dx.doi.org/10.1145/3020078.3021733

the 2011 revision of the C standard ('C11') [17, §7.17.3], and later incorporated into OpenCL [19, §3.3.4]. The exact guarantees provided by these operations are specified by the language's *memory consistency model* (MCM); the rough idea is that while SC forbids *all* reorderings, acquire loads cannot be executed *later*, release stores cannot be executed *earlier*, and relaxed accesses can be moved freely. We show that C11's acquire/release and relaxed consistency can be implemented using fewer dependencies than SC, and hence offer the potential for more efficient scheduling.

Unfortunately, weak atomics are notoriously hard to implement correctly. A failure to anticipate their complex and counterintuitive behaviours has been the root cause of bugs in compilers [28], language specifications [4], and vendor-endorsed programming guides [2]. To build confidence that our work implements C11 atomics correctly, we use the Alloy model checker [18], first to debug our implementation during development, and then to verify automatically that any C11 program (with a bounded number of memory accesses) will be synthesised correctly.

We implement our approach in the LegUp HLS framework [6]. LegUp is an attractive starting point because it is open-source and already has some support for multi-threaded programs [7].

We evaluate our work via a case study: an application in which threads communicate via lock-free circular buffers. We show that using SC atomics yields a 2.5x speedup compared to locks, and that switching from SC atomics to weak atomics (where safe to do so) yields a further 1.5x speedup. Compared to an unsound implementation that omits locks and atomics altogether, our weak atomics incur only a 7% performance overhead and a 3% area overhead.

In summary,

- we show that LegUp cannot (in general) synthesise multi-threaded algorithms without relying on locks, because some instruction reorderings permitted by its scheduler can introduce erroneous behaviours (§2);
- we modify LegUp's scheduling algorithm to impose extra *intra-thread* dependencies on the atomics provided by the C11 standard, thus ensuring correct *inter-thread* communication (§4.2), and we show that a lock-free buffer implemented in this way is on average 2.5x faster than one that uses locks (§5);
- we further modify the scheduler to support *weak* atomics, also part of the C11 standard, which suffice for many algorithms despite requiring fewer dependencies (§4.3), and we show that using weak atomics instead of SC atomics in our lock-free buffer leads to a further 1.5x speedup (§5); and
- we confirm automatically, using the Alloy model checker, that our revised scheduler correctly implements SC and weak atomics as defined by the C11 standard (§4.4).

Experimental data, source code, and Alloy model files are available online [1].

2. MOTIVATING EXAMPLES

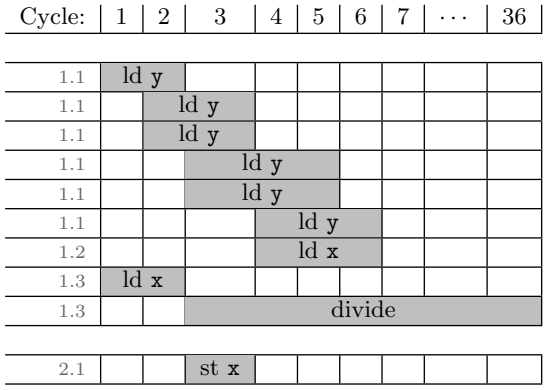
In this section, we provide two simple multi-threaded programs that can exhibit unexpected behaviours when compiled to hardware using LegUp, as a result of LegUp's relaxed scheduling constraints. In both cases, the unexpected behaviour only arises when particular instruction sequences

```
int x=0;
T1() {                 || T2() {
1.1 r0=x;              || 2.1 x=1;
1.2 r1=x;              ||     }
}
assert(r0 = 1 ⇒ r1 ≠ 0)
```

(a) A minimal violation of coherence.

```
int x=0; int y=0;
T1(int a) {            || T2() {
1.1 r0=y+y+y+y+y+y;    || 2.1 x=1;
1.2 r1=x;              ||     }
1.3 r2=x/a;            ||
}
assert(r1 = 1 ⇒ r2 ≠ 0)
```

(b) A coherence violation witnessed in LegUp (where thread `T1` is launched with `a` = 1).



(c) Schedules for threads `T1` (top) and `T2` (bottom).

Figure 1: Violating *coherence* in LegUp.

are carefully contrived, but we argue that similar sequences could easily occur in 'realistic' programs too. We emphasise that the unexpected behaviours discussed in this section do not mean that LegUp's scheduler is *wrong*, because the programs are racy and are hence technically illegal. However, if these programs were rewritten to use *atomics* (which *are* allowed to race), and LegUp were to implement atomics simply as ordinary non-atomic accesses, then it *would* be wrong.

Coherence.

A multi-threaded program conforms to *sequential consistency* (SC) if all memory accesses occur instantaneously and in the same order as the corresponding instructions in each thread [20]. One of the simplest violations of SC is a *coherence* violation [24, §8], as illustrated in Fig. 1a. The variable `x`, initially zero, is shared between two threads, `T1` and `T2`. A coherence violation occurs when the first load (line 1.1) observes `x`'s new value but the second load (line 1.2) observes `x`'s old value. This is detected by the failure of the final-state assertion.

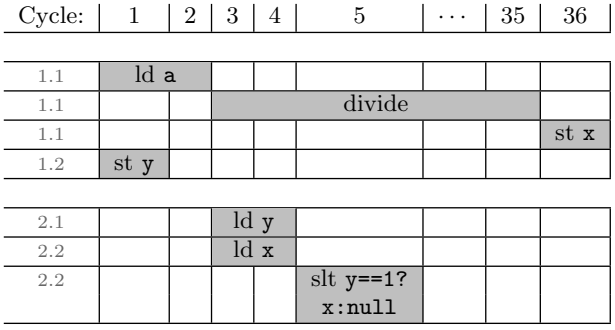
We could not observe this particular coherence violation in LegUp-generated circuitry, but we *could* observe a coherence violation by first making some innocuous transformations to the source code, as shown in Fig. 1b. These involve dividing one of the loaded values by a variable that is set to 1 at run-time (so the compiler cannot optimise it away), and

```
int x=0; int y=0;
T1() {                 || T2() {
1.1 x=1;               || 2.1 if(y==1)
1.2 y=1;               || 2.2   r0=x;
}                      || }
assert(r0 ≠ 0)
```

(a) A minimal violation of message-passing.

```
int x=0; int y=0;
T1(int a) {            || T2() {
1.1 x=a/3;             || 2.1 if(y==1)
1.2 y=1;               || 2.2   r0=x;
}                      || }
assert(r0 ≠ 0)
```

(b) A message-passing violation witnessed in LegUp (where thread `T1` is launched with `a` = 3).



(c) Schedules for threads `T1` (top) and `T2` (bottom).

Figure 2: Violating *message-passing* in LegUp.

inserting extra loads of a second shared location, `y`. These transformations result in LegUp finding the schedule shown in Fig. 1c.[1] Because of the high latency of the division operation, LegUp seeks to schedule the second read of `x` as early as possible. It determines that line 1.3 depends neither on line 1.2 (there is only a read-after-read (RAR) dependency on `x`) nor on line 1.1, and hence can be executed first in its thread. The repeated reads of `y` cause a delay between the two reads of `x`, and it is during this gap that thread `T2` updates `x`. In the main thread, threads `T1` and `T2` are forked successively, which offsets the starts of their respective executions by two cycles.

Message-passing.

Another example of an SC violation is illustrated by a failure of the *message-passing* paradigm [24, §3], which is illustrated in Fig. 2a. This example involves two shared locations, `x` and `y`, where `x` represents the message being passed from thread `T1` to thread `T2`, and `y` is used as a 'ready' flag. A message-passing violation occurs if `T2` observes that `y` has been set (line 2.1) but then goes on to observe that `x` is still zero (line 2.2).

As before, some innocuous code transformations are required to coax LegUp into revealing this behaviour, as shown in Fig. 2b. This time, we simply arrange that the value being stored to `x` is obtained by a division operation. As shown in the resultant schedule (Fig. 2c), this high-latency operation delays the store to `x`. Because lines 1.1 and 1.2 are deemed independent, the schedule permits them to execute simultaneously, and the result is that `y` is written first. In the reading thread (`T2`), LegUp schedules both loads simultaneously, having used if-conversion [23] to replace the control flow with predicated statements (`slt`). By launching the reading thread two clock cycles after the writing thread, we can observe the new value of `y` but the old value of `x` – a violation of message passing.

[1]The schedule is constrained by dual-ported memory access.

3. BACKGROUND

We now summarise existing HLS support for concurrent programming (§3.1), and introduce the C11 MCM (§3.2).

3.1 High-level synthesis

Several HLS tools only accept sequential input, deriving parallelisation opportunities either automatically (e.g. ROCCC [27], LegUp [6]) or with the aid of synthesis directives (e.g. Vivado HLS [31]). Other tools accept multi-threaded input but only allow threads to synchronise via locks (e.g. Kiwi [14]) or via execution barriers (e.g. SDAccel [30]). Some HLS tools also support the OpenMP programming standard, which defines an `atomic` directive that enables lock-free programming. Leow et al. [21] transform OpenMP to Handel-C for hardware synthesis and Cilardo et al. [8] generate heterogeneous hardware/software systems with OpenMP. Neither of these works support the explicit multi-threading constructs defined by the Pthreads standard, so a direct comparison with the present work is difficult. Altera's SDK for OpenCL [3] supports lock-free programming via atomics [26], though the commercial nature of the tool makes it difficult to ascertain exactly how these operations are implemented. LEAP facilitates parallel memory access through its provision of memory hierarchies that potentially can be shared among Pthreads in a lock-free manner [32].

The most important point of comparison between the tools reviewed above and the present work is that we are the first to synthesise hardware from software that features *weak* atomics (as defined by C11 [17] and OpenCL 2.x [19]). Efficient implementations of weak atomics have been extensively studied in the conventional processor domain, and they have been shown to yield *average, whole-program* speedups of 1.13x on x86 (Core i7) CPUs [25, Fig. 5] over their SC counterparts. Our circular buffer case study suggests that on FPGAs, weak atomics can yield a 1.5x average speedup.

Finally, Huang et al. [16] and Cong et al. [9] have shown that compiler optimisations can affect the quality of HLS-generated hardware. Our work shows that in a multi-threaded context, some optimisations (as manifested through relaxed scheduling constraints) can even be unsound.

3.1.1 HLS Scheduling

An HLS front-end converts source code into a *control/data flow graph* (CDFG) [11]. A CDFG is a directed graph where each vertex is a basic block (BB) and each edge represents a control-flow path. Each BB is a data-flow graph (DFG) with operations as vertices (V_{op}) and dependencies as edges ($E_{\text{d}} \subseteq V_{\text{op}} \times V_{\text{op}}$). Scheduling determines the start and end cycles of each operation in a CDFG, taking into account the control-flow and data dependencies as well as additional constraints such as latency and resources. Scheduling is performed alongside the allocation of resources and the binding

of operations and memory locations to these resources [11].

One of the most common scheduling techniques, used by Vivado HLS [31] and LegUp [6], expresses a CDFG schedule as a solution to a *system of difference constraints* (SDC) [10]. Various optimisations, such as as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) scheduling, can be obtained by reformulating the objective function. We focus in this work on the constraint that captures data dependencies, which is formulated as:

$$\forall (v, v') \in E_{\mathrm{d}} : end(v) - start(v') \leq 0.$$

That is, for every edge (v, v') where operation v' depends on v, the end of operation v must be scheduled before the start of operation v'.

3.1.2 LegUp

We utilise *LegUp*, an open-source HLS framework that enables applications to run on processors, as FPGA hardware accelerators, or both [6]. LegUp compiles an input C program to the LLVM Intermediate Representation (IR), which it analyses in order to generate a CDFG for SDC-based scheduling. To do this, LegUp firstly inserts control dependencies between BBs based on program order to ensure that only one BB is active at a time. These control dependencies eliminate any inter-BB instruction reordering or parallelism. Secondly, LegUp inserts data dependencies between instructions within a BB. LegUp insert two types of data dependencies (E_{d}): register (E_{reg}) and memory dependencies (E_{mem}), where

$$E_{\mathrm{d}} = E_{\mathrm{reg}} \cup E_{\mathrm{mem}}.$$

LegUp analyses register dependencies (E_{reg}) by identifying producer-consumer relationships between instructions, where data produced by an instruction is consumed by other instructions.

Our focus is on memory dependencies (E_{mem}), which hold between memory operations, $V_{\mathrm{mem}} \subseteq V_{\mathrm{op}}$. LegUp preserves read-after-write (RAW), write-after-write (WAW) and write-after-read (WAR) dependencies to aliasing memory locations, as shown below:

$$E_{\mathrm{mem}} = E_{\mathrm{LegUp}} \tag{1}$$

where

$$E_{\mathrm{LegUp}} = \{(v, v') \in V_{\mathrm{mem}} \times V_{\mathrm{mem}} \mid (v \in V_{\mathrm{st}} \vee v' \in V_{\mathrm{st}}) \wedge sb(v, v') \wedge sloc(v, v')\}$$

and $V_{\mathrm{st}} \subseteq V_{\mathrm{mem}}$ is the set of store operations (and elsewhere, $V_{\mathrm{ld}} \subseteq V_{\mathrm{mem}}$ is the set of load operations), sb is the 'sequenced before' relation (as determined from the program order), and $sloc$ is the 'same location' relation (as determined by an alias analysis tool). That is, E_{LegUp} contains every pair of aliasing memory operations v and v', where at least one is a store and where v is sequenced before v'.

LegUp's existing memory dependencies do not enforce ordering between memory instructions that have only read-after-read (RAR) dependencies, or that are non-aliasing. The omission of these orderings allows the potential for intra-BB out-of-order or overlapping execution of memory accesses. Such optimisations are *legal* in a single-threaded context and can lead to more efficient schedules.

LegUp's Pthread support allows multi-threaded C programs to be synthesised for FPGAs [7]. LegUp maps each thread to a CDFG, each of which is scheduled independently. Because these threads can be executed in parallel, LegUp provides locks to allow each thread mutually exclusive access to shared memory.

Instead of resorting to using locks, we embrace fine-grained concurrency by extending LegUp's Pthread flow to support atomics. By default, LegUp's scheduler does not ensure sufficient memory consistency for atomics. We augment LegUp's scheduler to add additional intra-thread ordering edges to E_{mem}; this ensures globally-synchronised memory behaviour, without locks.

3.2 The C11 memory consistency model

The 2011 revision of the C and C++ languages, 'C11', defines a suite of instructions called 'atomics', for loading from and storing to shared memory without the need for locks [17, §5.1.2.4, §7.17]. Co-existing with these atomics are ordinary (non-atomic) memory loads and stores. Each atomic can be assigned a *consistency mode* (also known as a *memory order*). The available modes include: *relaxed* (for loads and stores) *acquire* (for loads) *release* (for stores), and *SC* (for loads and stores). Non-SC atomics can be more efficient than SC atomics, but do not guarantee that all threads have a consistent view of the memory they share. Each consistency mode can be *roughly* understood by assuming that all threads *do* share a consistent view of memory, but that some instructions can take effect out of order:

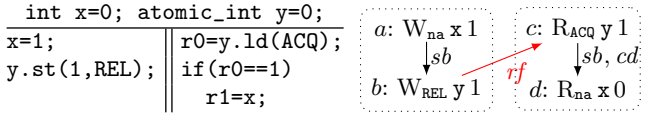
- an *atomic* load or store cannot be reordered with another atomic load or store that accesses the same location (this property is called *coherence*);
- a *relaxed atomic* load or store can be reordered anywhere within its thread (notwithstanding the other constraints);
- an *acquire atomic* load cannot be reordered with loads or stores that are sequenced after it in program order;
- a *release atomic* store cannot be reordered with loads or stores that are sequenced before it; and
- an *SC atomic* load or store cannot be reordered with any other load or store.

However, it is important to note that the explanations given above convey only a rough understanding. The official C11 standard defines the semantics of atomics not in terms of instruction reordering, but in terms of a detailed *memory consistency model* (MCM). The MCM specifies which complete executions of a program are allowed or forbidden. As a result of this discrepancy, some of the reorderings forbidden above are actually allowed under certain conditions.[2] This means that a program may actually exhibit more behaviours than a programmer following the rules above can anticipate.

The official semantics for C11 programs works by first mapping the program to a set of '*candidate*' traces [4]. This set of traces is obtained under the assumption that each load from a shared memory location can read a completely random value. In the second stage, candidate traces that exhibit inconsistent sequences of memory accesses among their events are rejected.

As an example of how this semantics works, consider the C11-style program in Figure 3a, and one of its candidate

[2]For instance, an acquire load *can* be reordered with a subsequent non-atomic load providing it is immediately preceded by another non-atomic load [12, §7.2].



(a) a program (b) a candidate trace

Figure 3: Example of C11 atomics.

traces (Figure 3b). We explain below why this particular candidate trace is deemed inconsistent. The trace contains four memory-related events (a, b, c, d), distributed between two threads as shown by the dotted rectangles. The store instructions give rise to write events (W) and the loads give rise to reads (R). Each event is tagged with the location it accesses (e.g., `x` or `y`), the value it reads or writes (e.g., 0 or 1), and whether it is non-atomic (`na`), atomic with consistency mode release (`REL`), or atomic with consistency mode acquire (`ACQ`). The *sequenced before* relation (sb) depicts the order of the instructions in the program, while the cd relation represents the control-flow dependency induced by the if-statement. The *reads-from* relation (rf) records that, in this particular trace, the read event c observes the 1 written by the write event b, and that the read event d (which has no incoming rf edge) observes the initial value, 0.

This trace is deemed inconsistent in C11 by the following reasoning. The rf arrow between the release and the acquire induces what is called 'release/acquire synchronisation' between the threads; we say that b *happens-before* c as a result. Taken together with the two sb arrows, we can further deduce that a happens-before d. C11 prescribes that reads must observe the most recent write in the happens-before relation, but d, which observes `x`'s initial value, violates this rule. Hence, the trace is disallowed.

4. METHOD

This section describes how we extend LegUp's Pthread flow [7] to support sequentially consistent (§4.2) and weakly consistent (§4.3) atomics.

As we discussed in §3.1.2, LegUp's MCM requires mutual exclusion (locks) to ensure safe access to shared memory in a multi-threaded context. We propose strengthening LegUp's MCM so that multi-threaded programs can synchronise using atomics rather than locks. We build on the LegUp framework, as it offers Pthread support and is open source, but our method is generally applicable to HLS tools that use SDC-based scheduling because we simply inject extra ordering edges as SDC data dependency constraints.

We compile atomic operations from the C11 standard with Clang 3.5 into LLVM IR. From the LLVM IR we can extract the atomicity of each memory operation, and the consistency mode of each atomic operation, and use this information to decide which ordering edges to inject into the scheduler. We focus on atomic loads and atomic stores in this paper, but our full implementation also includes fences [1]. We do not consider atomic read-modify-write instructions (such as compare-and-swap).

We propose three different strengthenings of LegUp's existing MCM, E_{mem}, which was discussed in §3.1.2. A naive approach, which gives the strongest possible MCM, is to adhere to strict program order and forbid any parallel memory access (§4.1). We also define an MCM that only imposes ordering on *atomic* memory accesses (§4.2) and a third MCM that relaxes some ordering for weak atomics (§4.3).

To help visualise the scheduling implications of the various MCMs, we provide a running example: a single thread that loads from four different memory locations. The third load is atomic with the acquire (`ACQ`) consistency mode; the rest are non-atomic (`na`). Each schedule is obtained using ASAP scheduling assuming an unlimited number of memory ports.

Cycle:	1	2
`r0=w;`	ld_{na} `w`	
`r1=x;`	ld_{na} `x`	
`r2=y.ld(ACQ);`	ld_{ACQ} `y`	
`r3=z;`	ld_{na} `z`	

The schedule above shows our running example implemented with LegUp's current MCM. LegUp treats atomic operations as regular memory operations and since these memory accesses do not alias, all four memory operations are free to be scheduled simultaneously.

4.1 Preserving SC semantics

A naive solution for correct program behaviour is to serialise all memory operations, regardless of any alias analysis.

This is achieved by redefining E_{mem} as follows:

$$E_{\text{mem}} = \{(v, v') \in V_{\text{mem}} \times V_{\text{mem}} \mid sb(v, v')\}. \quad (2)$$

E_{mem} now includes every pair of memory operations (v, v') where v is sequenced before v'. It overrides the memory dependencies generated by LegUp's existing MCM, E_{LegUp} (§3.1.2)

The schedule of our running example in this MCM is shown below.

Cycle:	1	2	3	4	5	6	7	8
`r0=w;`	ld_{na} `w`							
`r1=x;`			ld_{na} `x`					
`r2=y.ld(ACQ);`					ld_{ACQ} `y`			
`r3=z;`							ld_{na} `z`	

Because of the serialisation, this schedule cannot utilise more than one memory port for shared memory access. This stifles any parallelism offered by a multi-ported memory controller.

4.2 Exploring atomics

We now define an MCM that specifies ordering dependencies only for the *atomic* operations within each thread, $V_{\text{at}} \subseteq V_{\text{mem}}$. We treat all atomic operations as SC, regardless of the consistency mode specified in the program. To do this, we augment LegUp's original scheduling constraints with those in $E_{\text{at}\uparrow}$ and $E_{\text{at}\downarrow}$:

$$E_{\text{mem}} = E_{\text{LegUp}} \cup E_{\text{at}\uparrow} \cup E_{\text{at}\downarrow} \quad (3)$$

where

$$E_{\text{at}\uparrow} = \{(v', v) \in V_{\text{mem}} \times V_{\text{mem}} \mid sb(v', v) \wedge v \in V_{\text{at}}\}$$
$$E_{\text{at}\downarrow} = \{(v, v') \in V_{\text{mem}} \times V_{\text{mem}} \mid v \in V_{\text{at}} \wedge sb(v, v')\}.$$

$E_{\text{at}\uparrow}$ specifies that for every atomic operation v and every memory operation v' sequenced before v, there must exist an ordering edge from v' to v. $E_{\text{at}\downarrow}$ specifies that for every atomic operation v and every memory operation v' sequenced after v, there must exist an ordering edge from v to v'. The combination of these two constraints and LegUp's

existing MCM E_{LegUp} allows us to define an MCM that supports SC atomics.

The schedule of our running example when implemented in this MCM is shown below.

Cycle:	1	2	3	4	5	6
`r0=w;`	ld_{na} `w`					
`r1=x;`	ld_{na} `x`					
`r2=y.ld(ACQ);`			ld_{ACQ} `y`			
`r0=z;`					ld_{na} `z`	

The atomic load of `y` is constrained to happen after the loads of `w` and `x` (by $E_{\text{at}\uparrow}$) but before the load of `z` (by $E_{\text{at}\downarrow}$). Even though the atomic load uses the acquire consistency mode, this MCM treats it as a SC atomic load. The MCM definition in (3) is generally less restrictive than the one in (2) because ordering is only enforced with respect to atomics, but in the worst case, it is equivalent to (2) when all memory accesses are atomic ($V_{\text{at}} = V_{\text{mem}}$).

4.3 Exploiting weak atomics

In §4.2, we defined an MCM that treats all atomic operations as SC atomics. This approach is suboptimal whenever any atomics have consistency modes that are weaker than SC. In this subsection, we take advantage of the relaxations allowed for these weak atomics by injecting fewer ordering edges compared to SC atomics.

Let V_{sc}, V_{acq}, V_{rel}, and V_{rlx} be the sets of sequentially consistent, acquire, release and relaxed atomics, such that $V_{\text{sc}} \cup V_{\text{acq}} \cup V_{\text{rel}} \cup V_{\text{rlx}} = V_{\text{at}}$. We define a MCM that can support weak atomics to be the union of LegUp's existing MCM, E_{LegUp}, and the five sets of constraints given below:

$$E_{\text{mem}} = E_{\text{LegUp}} \cup E_{\text{sc}\downarrow} \cup E_{\text{sc}\uparrow} \cup E_{\text{acq}\downarrow} \cup E_{\text{rel}\uparrow} \cup E_{\text{RAR}} \quad (4)$$

where

$$\begin{aligned}
E_{\text{sc}\downarrow} &= \{(v, v') \in V_{\text{mem}} \times V_{\text{mem}} \mid v \in V_{\text{sc}} \wedge sb(v, v')\} \\
E_{\text{sc}\uparrow} &= \{(v', v) \in V_{\text{mem}} \times V_{\text{mem}} \mid sb(v', v) \wedge v \in V_{\text{sc}}\} \\
E_{\text{acq}\downarrow} &= \{(v, v') \in V_{\text{mem}} \times V_{\text{mem}} \mid v \in V_{\text{acq}} \wedge sb(v, v')\} \\
E_{\text{rel}\uparrow} &= \{(v', v) \in V_{\text{mem}} \times V_{\text{mem}} \mid sb(v', v) \wedge v \in V_{\text{rel}}\} \\
E_{\text{RAR}} &= \{(v, v') \in V_{\text{mem}} \times V_{\text{mem}} \mid sb(v, v') \wedge \\
&\qquad v \in V_{\text{at}} \cap V_{\text{ld}} \wedge v' \in V_{\text{at}} \cap V_{\text{ld}} \wedge sloc(v, v')\}.
\end{aligned}$$

We define five rules to implement an MCM that exploits the performance benefits of weak atomics. $E_{\text{sc}\uparrow}$ and $E_{\text{sc}\downarrow}$ define the ordering dependencies for SC atomics, which are similar to $E_{\text{at}\uparrow}$ and $E_{\text{at}\downarrow}$ from §4.2, except that they only apply to SC atomics rather than all atomics. $E_{\text{acq}\downarrow}$ represents the ordering edges for acquire atomics: for every memory operation v' sequenced after an acquire atomic v, there must exist an ordering edge from v to v'. $E_{\text{rel}\uparrow}$ represents the ordering edges for release atomics: for every memory operation v' sequenced before a release atomic v, there must exist an ordering edge from v' to v. E_{RAR} enforces read-after-read dependencies for all atomics: we inject an ordering edge from v to v' whenever v is sequenced before v' and both load from the same memory location (*sloc*).

The schedule of our running example for this MCM is shown below.

Cycle:	1	2	3	4
`r0=w;`	ld_{na} `w`			
`r1=x;`	ld_{na} `x`			
`r2=y.ld(ACQ);`	ld_{ACQ} `y`			
`r3=z;`			ld_{na} `z`	

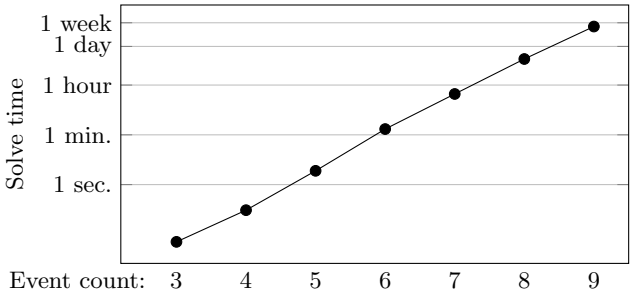


Figure 4: Solving time as the maximum number of events increases (y-axis is logarithmic).

Since the load of `y` is an acquire atomic, it must be completed before the load of `z` (by $E_{\text{acq}\downarrow}$), which is sequenced after it. However, the memory operations sequenced before the acquire load of `y` can be scheduled in parallel.

4.4 Ensuring correctness

Even though the scheduling constraints that we enforce are relatively straightforward, it is still challenging to justify that they are sufficient to rule out all executions deemed inconsistent by C11's MCM, because the specification of C11's MCM is so complex. Previous work has *proved* the correctness of implementations of C11's MCM both on CPUs [4] and on GPUs [28], but such proofs are laborious and fragile, and hence ill-suited to our prototype implementation.

Therefore, we turn to *lightweight* methods for verifying correctness. We employ the Alloy model checker [18] both to debug our implementation and to verify its correctness (up to a bound on the size of programs). Wickerson et al. [29] have previously used Alloy to check implementations of the C11 and OpenCL MCMs for several CPU and GPU architectures. Here, we port their work from the conventional processor domain to HLS.

Specifically, we use Alloy's constraint-solving abilities to search for a C11 trace T and a strict total order $\sqsubset_T$ over the events in T, such that

- T is *disallowed* by C11, but
- $v \sqsubset_T v'$ holds for all $(v, v') \in E_{\text{d}}$ – that is, $\sqsubset_T$ satisfies all of the scheduling constraints given in §4.3.

The $\sqsubset_T$ relation represents the order in which T's events occur at run-time. The existence of such a trace implies that the scheduling constraints need to be strengthened.

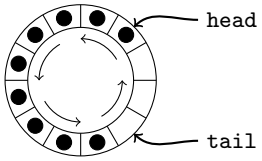
Figure 4 shows that Alloy's execution time increases exponentially with the upper bound on the number of events. The peformance figures were obtained on a machine with four 16-core 2.1 GHz AMD Opteron processors and 128 GB of RAM, and we used the Glucose SAT-solving backend. We were able to verify up to a maximum of 9 events. Although this bound appears small, many memory-related bugs can be revealed using even smaller programs [22]. We also confirmed that LegUp's original scheduling constraints are sufficient to avoid memory-related bugs in a single-threaded setting, again up to a 9-event bound.

5. EVALUATION

Thus far, our code examples have been relatively small, and designed to convey the problems of weak behaviour and

```
                atomic_int tail=0; head=0;
                 int arr[SIZE]; res[MSGS];
1.1  while(prod<MSGS) {          while(cons<MSGS) {          2.1
1.2   chead = head.ld(ACQ);       ctail = tail.ld(ACQ);      2.2
1.3   ctail = tail.ld(RLX);       chead = head.ld(RLX);      2.3
1.4   ntail = (ctail+1)%SIZE;     nhead = (chead+1)%SIZE;    2.4
1.5   if(ntail != chead){         if(ctail != chead){        2.5
1.6    arr[ctail] = prod           res[cons] = arr[chead];   2.6
1.7    tail.st(ntail,REL);         head.st(nhead,REL);      2.7
1.8    prod++;                     cons++;                  2.8
1.9   }                          }                          2.9
1.10 }                          }                          2.10
```

Figure 5: Acquire-Release semantics.



The `head` and `tail` pointers advance counterclockwise.

Figure 6: The circular buffer, diagrammatically.

demonstrate the potential of strengthening MCMs to implement atomics. 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 due to Hedström [15]. Data structures similar to this circular buffer are used in many real-time and memory-sensitive systems, and also appear in the Boost C++ library and the Linux kernel [5].

5.1 Case Study: SPSC Circular Buffer

Figure 5 shows the C-like code of a producer (on the left) and consumer (on the right) communicating via a circular buffer that is visualised in Figure 6. The buffer consists of atomic `head` and `tail` pointers, a buffer array (`arr`) and a result array (`res`). The producer only adds tasks and the consumer only removes tasks, as reflected by the store to `arr` (line 1.6) and the load from `arr` (line 2.6). The producer and consumer first check that the buffer is not full (line 1.5) and not empty (line 2.5), respectively. Finally, the producer and consumer update the `tail` (line 1.7) and `head` (line 2.7) pointers respectively with their next values. These next `tail` (line 1.4) and `head` (line 2.4) values are computed by a modular increment of `SIZE` to create a counterclockwise update, as depicted in Figure 6. We fix the buffer size (`SIZE`) at 64 and the number of messages transmitted (`MSGS`) to be 256. In addition, each atomic load (`ld()`) and atomic store (`st()`) is assigned a weak consistency mode: either `ACQ` for acquire, `REL` for release, or `RLX` for relaxed.

Ensuring correctness.

Hedström explains in detail why each memory access does not require full SC [15]. Roughly speaking, the non-atomic stores to `arr` (by the producer in line 1.6) do not race with the non-atomic loads of `arr` (by the consumer in line 2.6) because they are always separated by a release/acquire pair on the `tail` or the `head` pointer. These pairs ensure correct message-passing behaviour. The `tail` pair (lines 1.7 and 2.2) ensures that the consumer always reads from the latest write of the producer. Similarly, the `head` pair (lines 2.7 and 1.2) ensures that the consumer completes the read from `arr` before the producer writes to `arr` again.

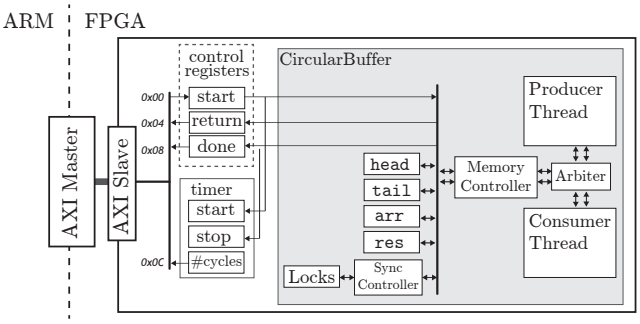


Figure 7: Architecture diagram.

Ensuring the correctness of any concurrent program in a weakly consistent setting is difficult because of the counter-intuitive behaviours allowed by a weak MCM, and testing is inconclusive because implementations of weakly consistent operations vary significantly between architectures. As such, to gain additional confidence in the correctness of this code, we turn to automatic verification. We use the CppMem tool [4] to confirm that the accesses of the shared non-atomic variable do not cause a race. Because CppMem does not support arrays, we replace `arr` with a scalar variable, and because CppMem's performance degrades rapidly with the number of events, we remove the while-loops. We give the actual code we verified online [1]. CppMem's result is of course weakened by the inclusion of these simplifications, but taken together with the informal argument for correctness given by Hedström, we obtain a reasonable degree of confidence in the program's correctness.

Implementation.

We map our buffer application to hardware via LegUp's pure hardware flow. We place-and-route our designs on a Xilinx Zynq 7000 (XC7Z020) with 53200 Look-up Tables (LUTs), 106400 registers, and 36 KB of block RAMs. Figure 7 shows the generated architecture. We synthesise each Pthread as a hardware accelerator, with global memory implemented on the FPGA either as registers or block RAMs. We accesses memory-mapped control registers from the ARM processor via an AXI slave connection, which we use to execute the accelerator system and extract both the verified results and the cycle counts from an on-board hardware timer. Shared memory access is protected by a memory controller. Although each thread has simultaneous dual-ported access to global memory, the memory controller can only perform two memory operations at a time. An arbiter ensures that only one thread is given access (to both ports) at a time. Also, LegUp's hardware locks are connected to the same memory controller via custom synchronisation logic.

5.2 Experiment Setup

We investigate the circular buffer on seven different design variations (as shown in Table 1): an unsound version, three lock-based versions, and three lock-free versions. Four of the seven are implemented with LegUp's pre-existing MCM and three are implemented with the MCMs discussed in §4.

The first version, Unsound, uses neither atomics nor locks. Although the results obtained from this implementation were verified to be correct experimentally, its correctness is coincidental and fragile. Small changes to the code, similar to those discussed in §2, could lead to incorrect results. We

Table 1: Design points. The last column gives the latency of one consume step and one produce step.

Short name	Description	MCM	Locks?	Lat.
Unsound	no atomics/locks (baseline)	orig	✗	10
OMP-criticals	OpenMP critical sections	orig	✓	14
Mutexes	Pthread mutexes	orig	✓	26
OMP-atomics	OpenMP atomics	orig	✓	41
SC	sequential consistency	§4.1	✗	17
SC atomics	sequentially consistent atomics	§4.2	✗	17
Weak atomics	weakly consistent atomics	§4.3	✗	12

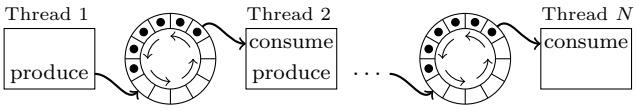


Figure 8: Chaining experiment (for N from 2 to 17).

treat this implementation as an upper bound for the circular buffer's performance.

OMP-criticals is implemented by wrapping the buffer's entire loop body with `#pragma omp critical`. LegUp implements these critical sections using a single global lock. Mutexes is similar, but in experiments that feature more than one circular buffer, each buffer is protected by its own mutex. This reduces the global contention to a single lock as can happen with OMP-criticals. OMP-atomics is implemented by wrapping each atomic instruction in a single-statement critical section. This is equivalent to OpenMP's `#pragma omp atomic`. Although this implementation is lock-free in source code, LegUp actually implements OpenMP atomics using a single global lock.

Of the lock-free designs, SC is the strictest because it serialises all memory operations regardless of their atomicity or consistency mode (see §4.1). SC atomics implements the MCM from §4.2 that takes into account the atomicity of memory operations but ignores the consistency mode of atomic operations. Weak atomics implements the MCM from §4.3 that considers the consistency mode of atomics.

We conduct two experiments on the circular buffer: chaining and bursting. Figure 8 shows the setup of the chaining experiment. The producer thread sends 256 messages across a chain of repeater threads to a consumer thread that verifies the results. Each repeater thread consumes from one buffer and immediately produces the same data to the next buffer in the chain. Table 1 provides the schedule latency of each repeater thread. In the chaining experiment, we observe the performance of our implementations as the number of repeater threads increases. In the bursting experiment, we increase the number of messages transmitted per transaction. We set up this experiment with three threads: one producer, one repeater, and one consumer. By increasing the number of messages per transaction, we increase the number of non-atomic memory accesses to the `arr` array. By doing so, we can investigate the relationship between the ratio of atomic accesses and the performance of our design points.

5.3 Results: Throughput

Figure 9 shows the throughput of the chaining and bursting experiments for all seven design points. For the chaining experiment, the overall throughput deteriorates with the increase in threads by an average of 20x across all design points. This can be explained by the fact that the arbitration cost to access shared memory is increasing with increase in threads. In the burst experiment, the overall throughput improves with the increase in elements per transaction by an average of 5x. By transmitting more messages per transaction, we decrease the overall synchronisation cost required to transfer the same amount of data across threads.

In both experiments, the Unsound implementation has the best throughput, which can be attributed to this design point having the shortest schedule latency (as seen in Table 1). All three lock-based implementations have large throughput overheads compared to the upper-bound performance of Unsound. This can be explained by the longer schedule latencies (Table 1). For each mutually-exclusive access, LegUp performs a pair of function calls to lock and unlock a hardware lock. OMP-criticals has one pair, Mutexes has two pairs, and OMP-atomics has six pairs of these function calls per repeater thread. These calls introduce schedule delays and additional memory dependencies. These delays affect OMP-atomics, which has an average overhead of 28x (chaining) and 10x (bursting) compared to the Unsound implementation.

In the chaining experiment, Mutexes outperforms OMP-criticals for four threads or more. As we increase the number of threads, Mutexes tends to have less overhead than OMP-criticals because it distributes the contention between multiple hardware locks, whereas OMP-criticals relies on a single lock. On average, the best lock-based implementations have performance overheads of 4x (chaining) and 3.5x (bursting) compared to Unsound.

Of the three lock-free implementations, SC has the largest performance overhead for both experiments, with 1.7x on average (chaining and bursting) compared to Unsound. Although SC exploits neither the atomicity nor the consistency modes of memory operations, it still outperforms the best lock-based implementations by 2.5x (chaining) and 2x (bursting). This suggests that dealing with memory consistency as an intra-thread scheduling problem is better than incurring the overhead of locks.

SC atomics is, in the worst-case, as restrictive as SC, and we see this behaviour in the chaining experiment, since both of these implementations have the same schedule latency (Table 1). In the burst experiment, we see that SC atomics performs better than SC when there are more elements per transaction. This corresponds to more non-atomic memory operations. The fewer atomic memory operation there are, the better SC atomics performs compared to SC. SC atomics performs up to 1.5x faster than SC in the burst experiment.

Finally, Weak atomics only has a performance overhead of 1.04x (chaining) and 1.15x (bursting) compared to Unsound. We are able to recover most of the performance of the Unsound implementation, while guaranteeing correct behaviour. Compared to SC atomics, Weak atomics is on average 1.6x faster (chaining) or 1.2x faster (bursting). As we increase the number of elements per transaction, the SC atomics throughput approaches that of Weak atomics, because the increased potential for parallelising the non-atomic accesses dominates any difference in the treatment of atomic accesses between the two implementations.

5.4 Results: Resources

Figure 10 shows LUT utilisation for the chaining and bursting experiments. We see an increase in LUT utilisa-

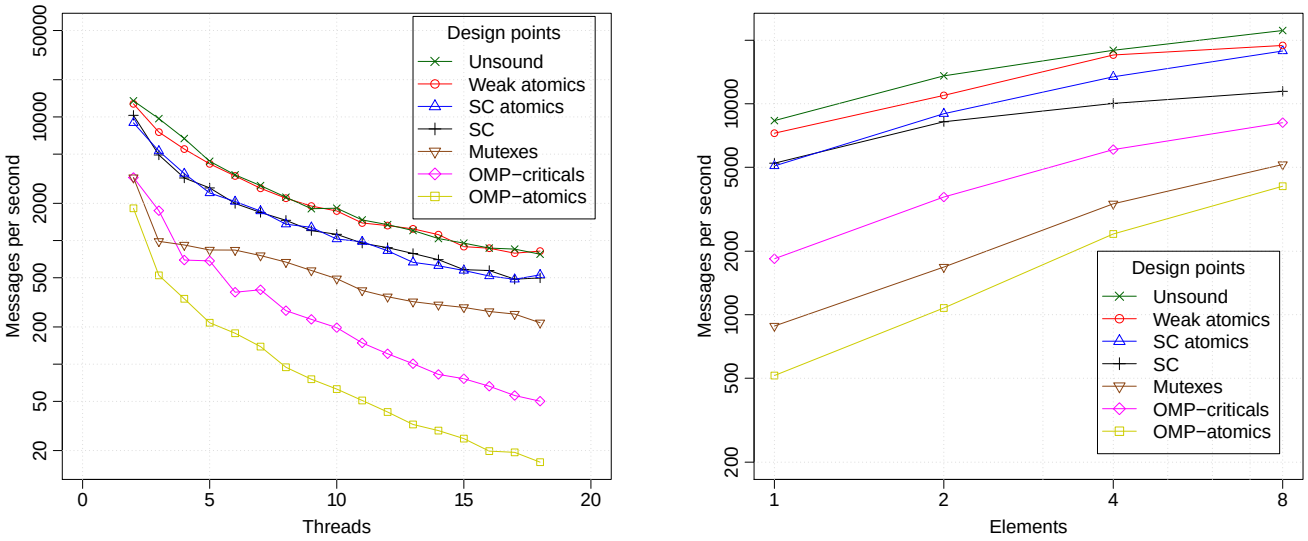


Figure 9: Throughput for the chaining experiment (left) and the bursting experiment (right).

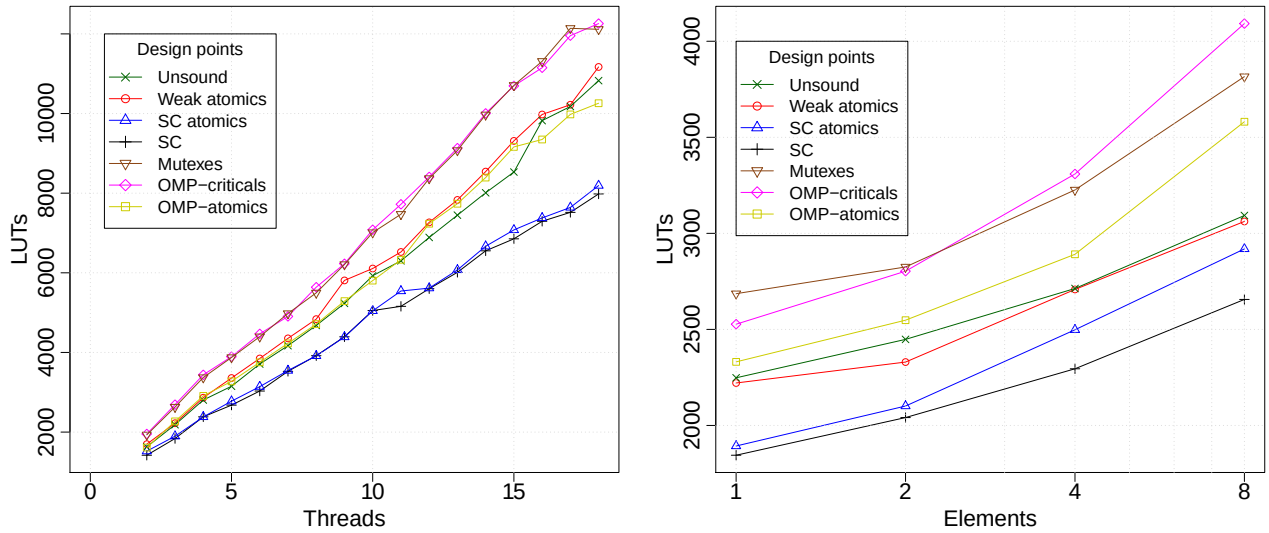


Figure 10: LUT utilisation for the chaining experiment (left) and the bursting experiment (right)

tion with the increase in threads and elements per transaction across all design points. At maximum LUT utilisation, we fill 22% of the FPGA fabric. OMP-criticals and Mutexes have the highest LUT utilisation. This can be attributed to both of these implementations requiring the synchronisation controller and hardware locks. SC and SC atomics have the smallest LUT utilisations, which can be attributed to them using only one memory port per thread due to serialisation (information we extract from LegUp's binding reports).

In the chaining experiment, we see that both the Unsound and the Weak atomics implementations use two memory ports per thread, resulting in their LUT utilisations being similar. For OMP-atomics, LUT utilisation lies between the SC and the Mutexes implementations. This may be because OMP-atomics requires the synchronisation controller and hardware locks (like Mutexes) but only uses one memory port per thread (like SC).

As we introduce more non-atomic memory accesses in the bursting experiment, some implementations can parallelise their intra-thread memory accesses and hence exploit the second memory port provided by LegUp. This can explain the rise in LUT utilisation that is particularly noticeable for SC atomics and OMP-atomics.

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-

ing weakly consistent atomics on FPGAs. Previous work on implementing weak atomics has concentrated on mapping C to processor-specific assembly code [4], [28]; our work shows how HLS can compile weak atomics directly to hardware.

In the future, we hope to extend our approach beyond loads and stores to handle compound atomic operations (such as compare-and-swap), and thus enable a larger class of lock-free programs to be synthesised into hardware.

Acknowledgements

We thank David Thomas and Jason Anderson for helpful discussions. The support of the EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS, grant reference EP/L016796/1), EPSRC grants EP/I020357/1, EP/K034448/1 and EP/K015168/1, the Royal Academy of Engineering, and Imagination Technologies is gratefully acknowledged.

REFERENCES

[1] Supplementary material is available on Zenodo, doi.org/10.5281/zenodo.200339, and GitHub, github.com/nadeshr/weak_atomics_FPGA17.

[2] J. Alglave, M. Batty, A. F. Donaldson, *et al.*, "GPU concurrency: weak behaviours and programming assumptions," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[3] Altera, *Altera SDK for OpenCL (2016.05.02)*, 2016.

[4] M. Batty, S. Owens, S. Sarkar, *et al.*, "Mathematizing C++ concurrency," in *Principles of Programming Languages (POPL)*, 2011.

[5] T. Blechmann, "Lock-free single-producer/single-consumer ringbuffer," bit.ly/2dbqFq1, 2013.

[6] A. Canis, J. Choi, M. Aldham, *et al.*, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Field-Programmable Gate Arrays (FPGA)*, 2011.

[7] J. Choi, S. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for FPGAs," in *Field-Programmable Technology (FPT)*, 2013.

[8] A. Cilardo, L. Gallo, A. Mazzeo, *et al.*, "Efficient and scalable OpenMP-based system-level design," in *Design, Automation & Test in Europe (DATE)*, 2013.

[9] J. Cong, B. Liu, R. Prabhakar, *et al.*, "A study on the impact of compiler optimizations on high-level synthesis," in *Languages and Compilers for Parallel Computing (LCPC)*, 2012.

[10] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *Design Automation Conference (DAC)*, 2006.

[11] P. Coussy, D. D. Gajski, M. Meredith, *et al.*, "An introduction to high-level synthesis," *IEEE Design and Test of Computers*, vol. 26, no. 4, 2009.

[12] M. Dodds, M. Batty, and A. Gotsman, "Compositional verification of relaxed-memory program transformations," Under review, 2016, bit.ly/2bmY7wI.

[13] V. Gramoli, "More than you ever wanted to know about synchronization," in *Principles and Practice of Parallel Programming (PPoPP)*, 2015.

[14] D. Greaves and S. Singh, "Kiwi: synthesis of FPGA circuits from parallel programs," in *Field-Programmable Custom Computing Machines (FCCM)*, 2008.

[15] K. Hedström, "Lock-free single-producer-single-consumer circular queue," bit.ly/2dbr8IK, 2014.

[16] Q. Huang, R. Lian, A. Canis, *et al.*, "The effect of compiler optimizations on high-level synthesis for FPGAs," in *Field-Programmable Custom Computing Machines (FCCM)*, 2013.

[17] ISO/IEC, *Programming languages – C*. International standard 9899:2011, 2011.

[18] D. Jackson, *Software Abstractions – Logic, Language, and Analysis*, Revised edition. MIT Press, 2012.

[19] Khronos Group, *The OpenCL Specification*. Version 2.0, 2013.

[20] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, 1979.

[21] Y. Y. Leow, C. Y. Ng, and W. F. Wong, "Generating hardware from OpenMP programs," in *Field-Programmable Technology (FPT)*, 2006.

[22] S. Mador-Haim, R. Alur, and M. M. K. Martin, "Litmus tests for comparing memory consistency models: how long do they need to be?" In *Design Automation Conference (DAC)*, 2011.

[23] S. A. Mahlke, R. E. Hank, R. A. Bringmann, *et al.*, "Characterizing the impact of predicated execution on branch prediction," in *Microarchitecture (MICRO)*, 1994.

[24] L. Maranget, S. Sarkar, and P. Sewell, "A tutorial introduction to the ARM and POWER relaxed memory models," bit.ly/2dbpUNu, 2012.

[25] N. Minh Lê, A. Pop, A. Cohen, *et al.*, "Correct and efficient work-stealing for weak memory models," in *Principles and Practice of Parallel Programming (PPoPP)*, 2013.

[26] N. Ramanathan, J. Wickerson, F. Winterstein, *et al.*, "A case for work stealing on FPGAs with OpenCL atomics," in *Field-Programmable Gate Arrays (FPGA)*, 2016.

[27] J. Villarreal, A. Park, W. Najjar, *et al.*, "Designing modular hardware accelerators in C with ROCCC 2.0," in *Field-Programmable Custom Computing Machines (FCCM)*, 2010.

[28] J. Wickerson, M. Batty, B. M. Beckmann, *et al.*, "Remote-scope promotion: clarified, rectified, and verified," in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[29] J. Wickerson, M. Batty, T. Sorensen, *et al.*, "Automatically comparing memory consistency models," in *Principles of Programming Languages (POPL)*, 2017.

[30] Xilinx, *SDAccel development environment - user guide (v2016.2)*, 2016.

[31] ——, *Vivado design suite user guide: high-level synthesis (v2016.2)*, 2016.

[32] H.-J. Yang, K. Fleming, M. Adler, *et al.*, "LEAP shared memories: automating the construction of FPGA coherent memories," in *Field-Programmable Custom Computing Machines (FCCM)*, 2014.