Scheduling Weakly Consistent C Concurrency for Reconfigurable Hardware

Nadesh Ramanathan, *Student Member, IEEE,* John Wickerson, *Member, IEEE,*
and George A. Constantinides *Senior Member, IEEE*
Imperial College London

***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 article 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. In addition, we show that we can support the pipelining of loops containing atomics by injecting further inter-iteration constraints. We implement our approach on two constraint-based scheduling HLS tools: LegUp 4.0 and LegUp 5.1. We extend both tools to support two memory models that are capable of synthesising atomics correctly. The first memory model only supports *sequentially consistent* (SC) atomics and the second supports *weakly consistent* ('weak') atomics as defined by the 2011 revision of the C standard. Weak atomics necessitate fewer constraints than SC atomics, but suffice for many multi-threaded algorithms. We confirm, via automatic model-checking, that we correctly implement the semantics in accordance with the C standard. A case study on a circular buffer suggests that on average circuits synthesised from programs that schedule atomics correctly can be 6x faster than an existing lock-based implementation of atomics, that weak atomics can yield a further 1.3x speedup, and that pipelining can yield a further 1.3x speedup.

***Index Terms*—High-Level Synthesis, HLS, Lock-Free Algorithms, Atomic Operations, FPGA.**

I. Introduction

In his comprehensive empirical study, Gramoli [1] demonstrates 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 article, 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 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 two instructions above may introduce the behaviour where `x` is assigned the latest value but `y` gets an old one.[1]

The implication of this is not that existing HLS tools are wrong; these optimisations can only introduce new behaviours when the code already exhibits a race condition, and races are deemed a programming error in C [2, §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 atomic operation in its own critical region, and ensure that the surrounding `lock()` and `unlock()` calls cannot be reordered. We show that this approach scales poorly and inhibits loop pipelining. 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 maintain a completely consistent view of shared memory, and memory accesses always occur in the order specified by the programmer [3]. Though simple for programmers to understand, SC is an expensive guarantee for language implementations to meet in the presence of optimisations by compilers (such as *constant propagation*, which can disrupt the order of memory accesses) and by architectures (such as *store buffering*, which can delay the propagation of writes to other threads).

In fact, many multi-threaded 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 the 2011 revision of the C standard ('C11') [2, §7.17.3], and later incorporated into OpenCL [4, §3.3.4]. The exact guarantees provided by these operations are specified by each

[1] Throughout this article, we use *thread* to refer both to software threads and to the hardware modules synthesised from them.

language's *memory consistency model*; 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. We also show how we can enable *loop pipelining* – an optimisation that is inhibited in the presence of locks but becomes available in our lock-free setting – by selectively imposing constraints between the memory operations in successive iterations of a loop.

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 [5], language specifications [6], and vendor-endorsed programming guides [7]. To build confidence that our work implements C11 atomics correctly, we use the Alloy model checker [8], 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 on two versions of the LegUp HLS framework [9]. We treat these two versions as separate tools for memory-related optimisations, as discussed in §V-B. We evaluate our approach in the context of both these tools via a case study: an application in which threads communicate via lock-free circular buffers. On average, we show that using SC atomics yields a 6x speedup compared to lock-based implementation of atomics, that switching from SC atomics to weak atomics (where safe to do so) yields a further 1.3x speedup, and that enabling loop pipelining of weak atomics can yield a further 1.3x speedup.

In summary,

- we show that traditional HLS schedulers cannot (in general) synthesise multi-threaded algorithms without relying on locks, because some instruction reorderings permitted by standard dependence-based schedulers that only consider aliasing memory dependencies can introduce erroneous behaviours, and we illustrate this using the open-source tool LegUp (§III);
- we extend the schedulers to impose extra *intra-thread* dependencies to support sequentially-consistent atomics provided by the C11 standard, thus ensuring correct *inter-thread* communication (§IV-A);
- we further modify the schedulers to support *weak* atomics, also part of the C11 standard, which suffice for many algorithms despite requiring fewer dependencies (§IV-B);
- we further extend the schedulers to support loop pipelining of atomics by injecting appropriate *inter-iteration* dependencies (§IV-C and IV-D); and
- we confirm automatically, using the Alloy model checker, that our revised scheduler correctly implements strong and weak atomics as defined by the C11 standard (§IV-E).

This article builds on results first presented in a conference paper [10]. The key additional results that this article reports are that support for loop pipelining has been added and evaluated via a new series of experiments, and that scheduling rules for strong and weak atomics have been implemented and evaluated in a second HLS tool (which shows that the speedups we obtain are not limited to a single tool). Experimental data, source code, and Alloy model files are available online [11].

II. Background

This section summarises existing HLS support for multi-threaded programming (§II-A), explains how HLS tools perform scheduling (§II-B), and introduces the C11 memory consistency model (§II-C).

A. High-level synthesis for multi-threaded programs

Several HLS tools only accept sequential input, deriving parallelisation opportunities either automatically (*e.g.* ROCCC [12]) or with the aid of synthesis directives (*e.g.* Vivado HLS [13]). Other tools accept multi-threaded input but only allow threads to synchronise via locks (*e.g.* LegUp [9] and Kiwi [14]) or via execution barriers (*e.g.* SDAccel [15]). Some HLS tools also support the OpenMP programming standard, which defines an `atomic` directive that enables lock-free programming. Leow *et al.* [16] transform OpenMP to Handel-C for hardware synthesis and Cilardo *et al.* [17] 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 [18] supports lock-free programming via SC atomics [19], 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 [20].

The most important point of comparison between the tools reviewed above and the present work is that this is the first to synthesise hardware from software that features *weak* atomics (as defined by C11 [2] and OpenCL 2.x [4]). Efficient implementations of weak atomics have been extensively studied in the conventional processor domain, with one study suggesting that they can yield average whole-program speedups of 1.13x on x86 (Core i7) CPUs [21, Fig. 5] over their SC counterparts. Sinclair *et al.* [22] suggest that weak atomics can achieve up to 1.5x speedup and 1.4x energy reduction on conventional GPU and integrated CPU-GPU systems. The authors also highlight that the benefits of weak atomics can vary between applications and inputs. Our circular buffer case study suggests that on FPGAs, weak atomics can yield a 1.3x average speedup. Our average speedup in this article differs from the speedup reported in our conference paper [10], since our results include long-latency experiments (division) and also a second HLS tool (LegUp 5.1). We also observe that our speedups in this article can vary between 1.2x to 1.6x depending on the experiment and tool.

Finally, Huang *et al.* [23] and Cong *et al.* [24] 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.

B. HLS Scheduling

An HLS front-end typically first converts source code into a *control/data flow graph* (CDFG) [25]. 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 ($E_{\text{d}} \subseteq V_{\text{op}} \times V_{\text{op}} \times \mathbb{N}$). Each edge is a triple comprising a source operation, a target operation, and a *dependence distance*, which is a natural number representing the number of loop iterations between those operations.

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 [25].

One of the most common scheduling techniques, used by Vivado HLS [13] and LegUp [9], expresses a CDFG schedule as a solution to a *system of difference constraints* (SDC) [26]. Various objectives, such as as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) scheduling, can be obtained by reformulating the objective function. Modulo scheduling [27], which is a well-known technique for loop pipelining, can also be implemented within an SDC framework [28].

In this work, we focus on the constraint that captures data dependencies, which is formulated as [29]:

$$\forall(v, v', \mathit{dist}) \in E_{\text{d}} : \mathit{end}(v) - \mathit{start}(v') \leq \mathit{II} \times \mathit{dist}.$$

That is, for every edge (v, v', dist) where operation v' depends on v, the number of cycles between the end of operation v and the start of operation v' must be at the least the loop initiation interval (II) multiplied by the loop dependence distance (dist), where II is the number of cycles between the initiations of two consecutive loop iterations. A dependence is *intra-iteration* when $\mathit{dist} = 0$, and is otherwise *inter-iteration*.

1) Memory dependencies for sequential programs: Memory dependencies (E_{mem}), which hold between memory operations, $V_{\text{mem}} \subseteq V_{\text{op}}$, are a subset of data dependencies ($E_{\text{mem}} \subseteq E_{\text{d}}$). C-based HLS tools perform alias analysis on an sequential C program and preserve read-after-write (RAW), write-after-write (WAW) and write-after-read (WAR) dependencies between aliasing memory locations.

These dependencies are preserved within a single loop iteration as follows:

$$E_{\text{intra-iter}} = \{(v, v', 0) \mid \mathit{sb}(v, v') \wedge \mathit{sloc}(v, v') \wedge (v \in V_{\text{st}} \vee v' \in V_{\text{st}})\} \quad (1)$$

where $V_{\text{st}} \subseteq V_{\text{mem}}$ is the set of store operations (and elsewhere, $V_{\text{ld}} \subseteq V_{\text{mem}}$ is the set of load operations), sb is the 'sequenced before' relation (as determined from the order of instructions in the original program), and sloc is the 'same location' relation (as determined by the alias analysis). $E_{\text{intra-iter}}$ expresses that there is a dependency from memory operation v to every memory operation v' that is sequenced after v in the same iteration, providing v and v' alias and at least one of them is a store.

In the absence of loop pipelining, we define E_{mem} as follows:

$$E_{\text{mem}} = E_{\text{intra-iter}} \cup E_{\text{nopipe}} \quad (2)$$

where E_{nopipe} inhibits any overlap between successive iterations:

$$E_{\text{nopipe}} = \{(v, v', 1) \mid v \in V_{\text{mem}} \wedge v' \in V_{\text{mem}}\}.$$

To obtain loop pipelining, we replace E_{mem} with $E^{\text{pipe}}_{\text{mem}}$, which imposes inter-iteration dependencies only between aliasing operations:

$$E^{\text{pipe}}_{\text{mem}} = E_{\text{intra-iter}} \cup E_{\text{inter-iter}} \quad (3)$$

where

$$E_{\text{inter-iter}} = \{(v, v', 1) \mid \mathit{sloc}(v, v') \wedge (v \in V_{\text{st}} \vee v' \in V_{\text{st}})\}.$$

$E_{\text{inter-iter}}$ expresses that there is a dependency from memory operation v to every memory operation v' that occurs in the next iteration, providing v and v' alias and at least one of them is a store.

The dependencies in (2) and (3) define memory consistency models (MCMs) that 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 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.

2) Synthesising multi-threaded programs: The general method that HLS tools adopt to synthesise multi-threaded programs is to map each thread to a CDFG, each of which is scheduled independently. Scheduling a thread is treated as scheduling a sequential C program which means that each thread's memory dependencies are defined by E_{mem} or $E^{\text{pipe}}_{\text{mem}}$ and that these dependencies are within the thread. We demonstrate, in §III, that E_{mem} and $E^{\text{pipe}}_{\text{mem}}$ are insufficient for the correct synthesis of multi-threaded programs that use atomic operations. One must either rely on locks, as LegUp does [30], or else strengthen the intra-thread constraints, which we show how to do in §IV.

Although locks guarantee correct memory behaviour, mutual exclusion of shared memory stifles the benefits of explicit parallelism provided by multi-threaded programs. In this work, we embrace fine-grained concurrency by strengthening E_{mem} and $E^{\text{pipe}}_{\text{mem}}$ to support atomics, instead of resorting to using locks.

C. 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 [2, §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).[2] Non-SC atomics can be more efficient than SC

[2]There exists also a little-used *consume* mode that is similar to *acquire*.

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* places no additional restrictions on reordering;
- 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 of the C11 memory consistency model. The official C11 standard defines the semantics of atomics not in terms of which individual instructions can or cannot be locally reordered, but instead in terms of which executions of the entire program are allowed. More specifically, it works by first mapping a given program to a set of 'candidate' executions under the assumption of a completely non-deterministic memory system, and then rejecting candidates that exhibit certain prohibited patterns of memory accesses [6]. As a result of this discrepancy, some of the reorderings forbidden above are actually allowed under certain conditions. (To give an arcane example: an acquire load *can* be reordered with a subsequent non-atomic load providing it is immediately preceded by another non-atomic load [31, §7.2].) This means that a program may actually exhibit more behaviours than a programmer following the rules above can anticipate.

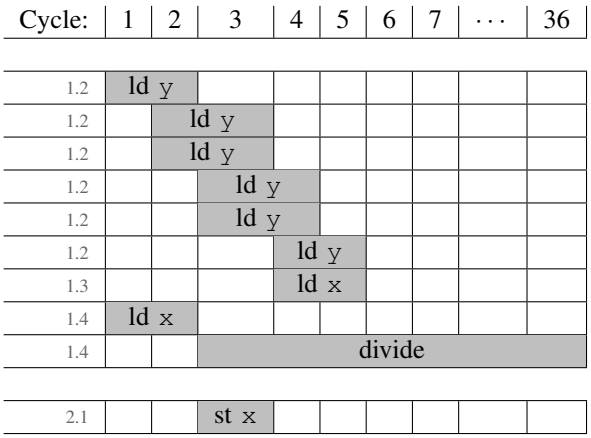
In order to ensure that our simple reordering rules are strong enough to enforce the complicated C11 memory consistency model, it is worthwhile to invest in formal verification – which we do in §IV-E.

```
atomic_int x=0;
T1() {                    ||  T2() {
1.1 int r0=0,r1=0;        ||  2.1 st(&x,1);
1.2 r0=ld(&x);            ||    }
1.3 r1=ld(&x);            ||
  }                       ||
assert(r0 = 1 ⇒ r1 ≠ 0)
```

(a) A minimal example of a coherence violation. The assertion failing indicates a coherence violation, where the second load of `x` (`r1`) happens before the first load of `x` (`r0`).

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

(b) A program that can exhibit a coherence violation when compiled using LegUp, if atomics are treated as non-atomics (thread T1 is launched with $a = 1$).



(c) Schedules for threads T1 (top) and T2 (bottom) that allow the program in (b) to exhibit a coherence violation.

Fig. 1. An example of how treating atomics as non-atomics can lead to *coherence* violations.

III. Motivating Examples

In this section, we provide two simple multi-threaded programs that could demonstrate unexpected behaviours as a result of the standard approach to high-level synthesis scheduling described as E_{mem} in (2). In both cases, the unexpected behaviour only arises when particular instruction sequences are carefully contrived, but we argue that similar sequences could easily occur in 'realistic' programs too.

To make our examples concrete, we present actual LegUp schedules demonstrating these behaviours occurring in practice (and make our source code available online [11]). However, our examples are relevant to any HLS tool that performs constraint-based scheduling on a per-thread basis in the absence of locks.

We emphasise that the unexpected behaviours discussed in this section do not mean that LegUp's scheduler is wrong, because LegUp does not claim to provide support for C11 atomics. Rather, we use these examples to demonstrate how the scheduling rules need to be altered to handle atomics correctly, and thereby avoid the problematic cases demonstrated in this section.

a) Coherence: A multi-threaded program conforms to *sequential consistency* (SC) if all memory accesses appear to occur instantaneously and in the same order as the corresponding instructions in each thread [3]. One of the simplest violations of SC is a *coherence* violation [32, §8], as illustrated in Fig. 1a. The atomic variable `x`, initially zero, is shared between two pthreads, `T1` and `T2`, that are synthesised as parallel hardware executing concurrently. A coherence violation occurs when the first load (Line 1.2) observes `x`'s new value but the second load (Line 1.3) observes `x`'s old value. This is detected by the failure of the final-state assertion. The reason for this violation of coherence is the absence of the read-after-read (RAR) edges in $E_{\text{intra-iter}}$ in (1).

We can observe a coherence violation in LegUp by first

making some innocuous transformations to the source code, as shown in Fig. 1b. We have replaced the atomic loads and stores (written `ld` and `st`) with their volatile counterparts, in order to simulate atomic operations being treated as unoptimised regular loads and stores. LegUp does not understand atomic variables and operations but can synthesise volatile variables, which ensures that no memory accesses are optimised away. The key transformation in Fig. 1b is to increase the priority of the second load of `x` compared to the first one in an ASAP scheduling context. We do so by feeding the second load's value into a division operation (whose denominator is set to 1 at run-time to avoid compiler optimisations) and also injecting extra loads of `y` that are part of an addition chain.

These transformations result in the schedule shown in Fig. 1c.[3] Because of the high latency of the division operation , the scheduler seeks to schedule the second read of `x` as early as possible. It determines that Line 1.4 depends neither on Line 1.3 (there is only a read-after-read (RAR) dependency on `x`) nor on Line 1.2, 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 (cycle 3) that thread `T2` updates `x`. The resultant execution shows that the second load of `x` reads stale data because the first load of `x` reads the latest value.

b) Message-passing: Another example of an SC violation is illustrated by a failure of the *message-passing* paradigm [32, §3], which is illustrated in Fig. 2a. This example involves two shared locations, `x` and `y`, where `x` represents a 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 2.3) but then goes on to observe that `x` is still 0. The reason for this violation is the absence of dependency edges between non-aliasing accesses in $E_{\text{intra-iter}}$ in (1).

As before, some innocuous code transformations are required to coax the ASAP scheduler into revealing this behaviour, as shown in Fig. 2b. As before, we replace the atomic variables and operations with their volatile counterparts, in order for the example to be synthesised by LegUp. 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`), both loads are scheduled simultaneously having used if-conversion [33] to replace the control flow with predicated statements (`slt`). By carefully launching the reading thread after the writing thread, we can observe the new value of `y` but the old value of `x` – a violation of message passing.

IV. Method

This section describes how we extend HLS scheduling to support sequentially consistent (SC) and weakly consistent C11 atomics, both in standard C synthesis and also in the loop pipelining context. We have implemented our method both via the Pthread flow of LegUp 4.0 and LegUp 5.1. 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 first compile C11 atomics into LLVM IR. From the LLVM IR, we can extract all memory operations and identify the atomic operations and their consistency modes. We use this information to inject the ordering edges into the scheduler.

In this work, we focus on atomic loads and stores and do not consider atomic read-modify-write (RMW) instructions (such as compare-and-swap). Realising atomic loads and stores only requires extending the scheduling constraints on LegUp and reusing the load and store primitives provided by LegUp. In contrast, atomic RMWs are not supported on LegUp and implementing them not only requires scheduling extensions but, more importantly, requires RTL changes to LegUp's underlying RTL backend to support this new primitive, thus we have reserved it for future work. We also can handle fences since implementing them only requires scheduling-level extension, similar to atomic loads and stores. We do not discuss

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

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

T1() {	T2() {
1.1 x=1;	2.1 int r0=0,r1=0;
1.2 st(&y,1);	2.2 r0=ld(&y);
}	2.3 if(r0==1) r1=x;
	}

$\texttt{assert}(\texttt{r0} = 1 \Rightarrow \texttt{r1} = 1)$

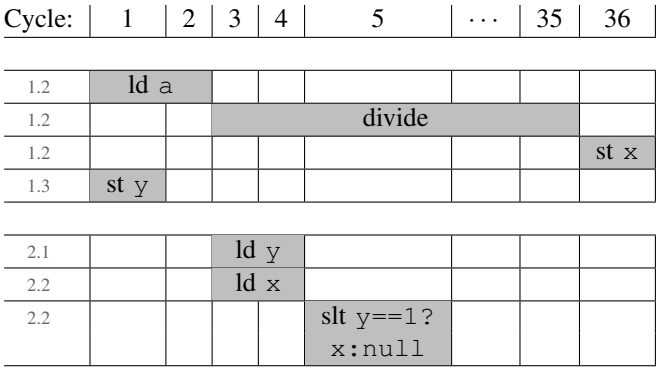
(a) A minimal example of a message-passing violation. The assertion failing indicates a message-passing violation, where the flag is set ($\texttt{r0} = 1$) but the data is stale ($\texttt{r1} = 1$).

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

T1(int a) {	T2() {
1.1 x=a/3;	2.1 int r0=0,r1=0;
1.2 y=1;	2.2 r0=y;
}	2.3 if(r0==1) r1=x;
	}

$\texttt{assert}(\texttt{r0} = 1 \Rightarrow \texttt{r1} = 1)$

(b) A program that can exhibit a message-passing violation when compiled using LegUp, if atomics are treated as non-atomics (thread `T1` is launched with $\texttt{a} = 3$).



(c) Schedules for threads `T1` (top) and `T2` (bottom) that allow the program in (b) to exhibit a message violation.

Fig. 2. An example of how treating atomics as non-atomics can lead to *message-passing* violations.

them in this article for brevity but we provide verification details of fences in our supplementary material [11].

We strengthen E_{mem} and $E_{\text{mem}}^{\text{pipe}}$, the MCMs discussed in §II-B1, to support atomic operations by defining two MCMs that implement SC atomics (§IV-A) and weak atomics (§IV-B). Our SC MCM implements all atomic operations as sequentially-consistent atomic operations, which is the default for C11 atomic operations when the consistency mode is left unspecified. This MCM represents the strongest-possible implementation of C atomics, and is the simplest to implement, since it requires the fewest additional rules.

The second MCM we implement considers the consistency mode of each atomic operation. This allows us to exploit further relaxations afforded by the C11 standard for non-SC atomics, potentially leading to better performance. This MCM requires more rules, which adds to the implementation complexity and increases the vulnerability to bugs – a risk that we mitigate using formal verification (§IV-E). We support relaxations for three consistency modes: acquire, release and relaxed atomics.[4]

Additionally, we extend both these MCMs to support loop pipelining by adding inter-iteration dependencies. The number of rules required to implement loop pipelining for weak atomics (§IV-D) is greater than for SC atomics (§IV-C).

To help visualise the scheduling implications of the various MCMs we propose, we provide a running example: a single thread that loads from three different memory locations. The second load is atomic with the acquire (`ACQ`) consistency mode; the rest are non-atomic (`na`). We assume that these three loads are the body of a loop that can be pipelined. The dark shade in the schedule represents the first iteration of a loop and the lighter shade represents the second iteration. We also assume ASAP scheduling and an unconstrained number of memory ports, for simplicity of exposition.

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

The schedule above shows our running example implemented using the existing E_{mem} MCM from §II-B1. The scheduler treats atomic operations as ordinary operations, and since these memory accesses do not alias, all three memory operations within a single iteration are free to be scheduled simultaneously. The schedule is the same for $E_{\text{mem}}^{\text{pipe}}$, because the three loads in the second iteration must execute after their respective aliasing loads from the first iteration, as specified by $E_{\text{inter-iter}}$.

A. Exploring SC atomics

We now define an MCM that injects additional intra-thread memory dependencies for the *atomic* operations within each thread, $V_{\text{at}} \subseteq V_{\text{mem}}$. In this MCM, all atomics are treated as SC atomics. By doing so, we provide a conservative but simple MCM that support atomics. To do this, we augment the original scheduling constraints (E_{mem}) with two additional rules, $E_{\text{at}\updownarrow}$ and $E_{\text{at}\downdownarrows}$, which prevent atomics from moving 'up' or 'down' in the schedule, respectively:

$$E_{\text{mem,SC}} = E_{\text{intra-iter}} \cup E_{\text{nopipe}} \cup E_{\text{at}\updownarrow} \cup E_{\text{at}\downdownarrows} \tag{4}$$

where

$$E_{\text{at}\updownarrow} = \{(v, v', 0) \mid sb(v, v') \wedge v \in V_{\text{at}}\}$$
$$E_{\text{at}\downdownarrows} = \{(v, v', 0) \mid sb(v, v') \wedge v' \in V_{\text{at}}\}.$$

$E_{\text{at}\updownarrow}$ 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' of dependence distance 0. $E_{\text{at}\downdownarrows}$ 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' of dependence distance 0. The combination of these two constraints with the original MCM $E_{\text{intra-iter}}$ allows us to define an MCM that supports C11 atomics. We treats all atomics as SC atomics by applying the same rules to the entire set of atomics V_{at}.

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

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12
`r1=x;`	ld_{na} x						ld_{na} x					
`r2=ld(&y,ACQ);`			ld_{ACQ} y						ld_{ACQ} y			
`r0=z;`					ld_{na} z						ld_{na} z	

Within an iteration, the atomic load of `y` is constrained to happen after the loads of `x` (by $E_{\text{at}\updownarrow}$) but before the load of `z` (by $E_{\text{at}\downdownarrows}$). Even though the atomic load uses the acquire consistency mode, this MCM treats it as a SC atomic load. The second iteration executes after the completion of the first, since this MCM does not support loop pipelining. Therefore, both the latency and initiation interval are equal to 6 for this MCM's schedule.

B. Exploiting weak atomics

Having defined a simple MCM that implements all C11 atomics as if they are SC, we now show how to take advantage of the relaxations allowed for weakly-consistent atomics, according to the C standard.

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 as follows:

$$\begin{aligned} E_{\text{mem,weak}} = {} & E_{\text{intra-iter}} \cup E_{\text{nopipe}} \cup \\ & E_{\text{sc}\updownarrow} \cup E_{\text{sc}\downdownarrows} \cup E_{\text{acq}\updownarrow} \cup E_{\text{rel}\downdownarrows} \cup E_{\text{RAR}} \end{aligned} \tag{5}$$

where

$$E_{\text{sc}\updownarrow} = \{(v, v', 0) \mid sb(v, v') \wedge v \in V_{\text{sc}}\}$$
$$E_{\text{sc}\downdownarrows} = \{(v, v', 0) \mid sb(v, v') \wedge v' \in V_{\text{sc}}\}$$
$$E_{\text{acq}\updownarrow} = \{(v, v', 0) \mid sb(v, v') \wedge v \in V_{\text{acq}}\}$$
$$E_{\text{rel}\downdownarrows} = \{(v, v', 0) \mid sb(v, v') \wedge v' \in V_{\text{rel}}\}$$
$$E_{\text{RAR}} = \{(v, v', 0) \mid sb(v, v') \wedge sloc(v, v') \wedge v \in V_{\text{at}} \cap V_{\text{ld}} \wedge v' \in V_{\text{at}} \cap V_{\text{ld}}\}.$$

We need five rules to implement an MCM that exploits the performance benefits of weak atomics, in contrast to two rules

[4] *Consume* atomics can be implemented as *acquire* atomics. *Acquire-release* atomics, that are only applicable to read-modify-writes and fences, can be implemented as *sequentially-consistent* atomics.

for the MCM in §IV-A. $E_{\text{sc}\uparrow}$ 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 §IV-A, except that they only apply to SC atomics rather than all atomics. $E_{\text{acq}\downarrow}$ imposes that acquire atomics cannot move 'down' in the schedule: for every memory operation v' sequenced after an acquire atomic v, there must exist an intra-iteration ordering edge from v to v'. $E_{\text{rel}\uparrow}$ imposes that release atomics cannot move 'up' in the schedule: for every memory operation v sequenced before a release atomic v', there must exist an intra-iteration ordering edge from v to v'. E_{RAR} enforces read-after-read (RAR) dependencies for all atomics: we inject an intra-iteration ordering edge from v to v' whenever v is sequenced before v' and both load from the same memory location ($sloc$). This rule differentiates relaxed atomics from non-atomic memory operations since non-atomic memory operations do not enforce RAR dependencies. It is needed to enforce C11's coherence rule (§II-C).

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

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

Since the load of y is an acquire atomic, it must execute 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. This MCM still does not support loop pipelining, so the second iteration must execute after the first iteration. The loop's latency and initiation interval are both equal to 4.

C. Pipelining SC atomics

In §IV-A, we defined an MCM that supports atomics in a non-pipelined setup. We now extend this MCM to ensure the correct execution of atomics in a pipelining context. We do so by injecting additional inter-iteration dependencies on top of the intra-iteration dependencies from (4). Similarly to the MCM in §IV-A, we treat all C11 atomics as SC atomics. By doing so, we can support pipelining using just one additional rule, as given below:

$$E^{\text{pipe}}_{\text{mem,SC}} = E_{\text{intra-iter}} \cup E_{\text{inter-iter}} \cup E_{\text{at}\downarrow} \cup E_{\text{at}\uparrow} \cup E_{\text{at-inter-iter}} \tag{6}$$

where

$$E_{\text{at-inter-iter}} = \{(v, v', 1) \mid v \in V_{\text{at}} \wedge v' \in V_{\text{mem}}\}.$$

$E_{\text{at-inter-iter}}$ defines the inter-iteration dependencies required for all atomics: for every atomic operation v and every memory operation v' that executes in the iteration after v, there must exist an inter-iteration edge from v to v' of distance 1.

The schedule for our running example for this MCM is shown below:

Cycle:	1	2	3	4	5	6	7	8	9	10
`r1=x;`	ld_{na} x				ld_{na} x					
`r2=ld(&y,ACQ);`			ld_{ACQ} y				ld_{ACQ} y			
`r0=z;`					ld_{na} z				ld_{na} z	

The first iteration's schedule is the same as in §IV-A, where the first atomic load of y must happen after first load of x and before the first load of z. The second load of x must happen after the first load of x (by $E_{\text{inter-iter}}$) and the first load of y (by $E_{\text{at-inter-iter}}$), but there is no rule that prohibits the first load of z and the second load of x happening in parallel. Therefore, the loop latency is still 6, as it is in §IV-A, but the initiation interval is now 4.

D. Pipelining weak atomics

We now define an MCM that enables loop pipelining for weak atomics by extending the weak MCM defined in §IV-B. We do so by defining four separate rules to handle SC, acquire, release and relaxed atomics in a loop pipelining context, as given below:

$$\begin{aligned} E^{\text{pipe}}_{\text{mem,weak}} = {} & E_{\text{intra-iter}} \cup E_{\text{inter-iter}} \cup E_{\text{sc}\downarrow} \cup E_{\text{sc}\uparrow} \cup \\ & E_{\text{acq}\downarrow} \cup E_{\text{rel}\uparrow} \cup E_{\text{RAR}} \cup \\ & E_{\text{sc-inter-iter}} \cup E_{\text{acq-inter-iter}} \cup \\ & E_{\text{rel-inter-iter}} \cup E_{\text{RAR-inter-iter}} \end{aligned} \tag{7}$$

where

$$\begin{aligned} E_{\text{sc-inter-iter}} &= \{(v, v', 1) \mid v \in V_{\text{sc}} \wedge v' \in V_{\text{mem}}\} \\ E_{\text{acq-inter-iter}} &= \{(v, v', 1) \mid v \in V_{\text{acq}} \wedge v' \in V_{\text{mem}}\} \\ E_{\text{rel-inter-iter}} &= \{(v, v', 1) \mid v' \in V_{\text{rel}} \wedge v \in V_{\text{mem}}\} \\ E_{\text{RAR-inter-iter}} &= \{(v, v', 1) \mid sloc(v, v') \wedge \\ &\qquad v \in V_{\text{at}} \cap V_{\text{ld}} \wedge v' \in V_{\text{at}} \cap V_{\text{ld}}\}. \end{aligned}$$

$E_{\text{sc-inter-iter}}$ and $E_{\text{acq-inter-iter}}$ are similar to $E_{\text{at-inter-iter}}$ but apply only to SC and acquire atomics respectively, where for every v that is either SC or acquire atomic and memory operation v' that executes in the iteration after v, there must exist an inter-iteration edge from v to v' of dependence distance 1. $E_{\text{rel-inter-iter}}$ only applies to release atomics, where for every memory operation v and release atomic v' that executes in the iteration after v, there must exist an inter-iteration edge from v to v' of dependence distance 1. $E_{\text{RAR-inter-iter}}$ applies to all atomics including relaxed atomics, where for every atomic load v and atomic load v' that executes in the iteration after v and to the same location ($sloc$), there must exist an inter-iteration edge from v to v' of dependence distance 1.

The schedule for our running example for this MCM that support pipelining weak atomics is shown below:

Cycle:	1	2	3	4	5	6
`r1=x;`	ld_{na} x		ld_{na} x			
`r2=ld(&y,ACQ);`	ld_{ACQ} y		ld_{ACQ} y			
`r3=z;`			ld_{na} z		ld_{na} z	

The first iteration's schedule is the same as in §IV-B, where the first loads of x and y can happen in parallel but the first load of z must happen after the first load of y (by $E_{\text{acq}\downarrow}$). The second load of x must happen after the first load of x (by $E_{\text{inter-iter}}$) and the first load of y (by $E_{\text{acq-inter-iter}}$), and can therefore start on the third cycle. This leads to a loop latency of 4 and an initiation interval of 2.

E. 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 [6] and on GPUs [5], 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 [8] both to debug our implementation and to verify its correctness (up to a bound on the size of programs). Alloy is a mature and well-supported tool whose input language (based on relational algebra) closely matches the style in which most MCMs are written. It has previously been used by Wickerson *et al.* [34] 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 execution T and a strict total order $\sqsubset_T$ over the operations in T, such that

1) T is *disallowed* by C11, but
2) $\sqsubset_T$ is consistent with all of the scheduling constraints given in §IV-B.

The $\sqsubset_T$ relation represents the order in which T's operations occur at run-time. Other relations in the execution T include '$sloc$' (for when two events access the same location), 'sb' (for when one event is sequenced before another), '$site$' (for when two events originate from the same loop iteration), and '$nite$' (for when one event originates from the next iteration after another event). Our setup supports pipelining by having intra-iteration constraints restricted to $site$-related events, and inter-iteration constraints restricted to $nite$-related events.[5] The existence of an execution that satisfies conditions 1 and 2 above would imply that the scheduling constraints need to be strengthened.

Alloy was able to verify that for all executions with up to nine operations, conditions 1 and 2 cannot be simultaneously satisfied. Because this represents an exhaustive search of all executions of *all* programs, it guarantees that our scheduling constraints are sufficient to rule out any memory-related bug that can be expressed using no more than nine memory operations. Though our verification result is bounded, it is still quite a strong result because many common memory-related bugs can be minimised to even smaller programs, typically comprising between four and six operations [35].

To give an indication of the computational effort involved in establishing this verification result, Fig. 3 shows that Alloy's verification time increases exponentially with the upper bound on the number of operations. 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 also confirmed that the original scheduling constraints are sufficient to avoid memory-related bugs in a single-threaded setting, again up to a 9-operation bound.

Although the constraints we have presented in this section have been shown to be correct, it is interesting to note that Alloy was able to detect a bug in an earlier version of the constraints, which we had lifted directly from the reordering rules for C11 atomics listed on the widely-used cppreference.com website.[6] Alloy found a program that could be scheduled to reveal an illegal behaviour. The earlier constraints forbade acquire loads to be reordered with any subsequent loads; the revised constraints forbid reordering with any subsequent loads *or stores*. We traced this bug back to the cppreference.com website itself, and it has now been fixed [36].

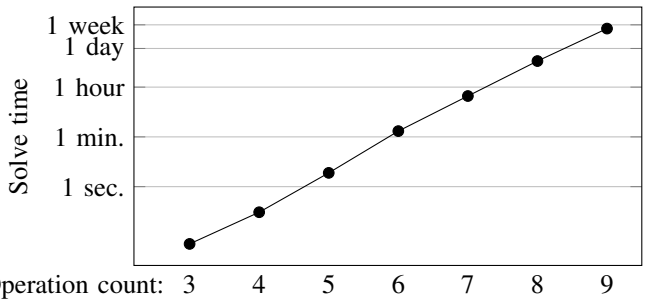


Fig. 3. Solving time to verify our MCMs in §IV up to a bounded number of operations.

TABLE I
DESIGN POINTS. THE FIRST FIVE REPRESENT EXISTING HLS SOLUTIONS; THE LAST FOUR ARE THE SOLUTIONS PROPOSED IN THIS WORK.

Short name	Description	MCM	Ref.
Unsound	constraints treat atomics as ordinary accesses (baseline)	E_{mem}	§II-B1
Pipelined unsound	like *Unsound*, but pipelined	$E^{\mathrm{pipe}}_{\mathrm{mem}}$	§II-B1
Atomic locks	like *Unsound*, but with a lock around each atomic	E_{mem}	
SC atomics	constraints treat all atomics as if they are SC	$E_{\mathrm{mem,SC}}$	§IV-A
Weak atomics	constraints sensitive to consistency modes of atomics	$E_{\mathrm{mem,weak}}$	§IV-B
Pipelined SC atomics	like *SC atomics*, but pipelined	$E^{\mathrm{pipe}}_{\mathrm{mem,SC}}$	§IV-C
Pipelined weak atomics	like *Weak atomics*, but pipelined	$E^{\mathrm{pipe}}_{\mathrm{mem,weak}}$	§IV-D

V. EVALUATION

We evaluate our method on two applications/data structures: a two-threaded message-passing (MP) channel and a single-producer-single-consumer (SPSC) circular buffer. We use message-passing as a pilot experiment to check the sanity of our implementations (§V-C), and SPSC circular buffers as our main case study (§V-D).

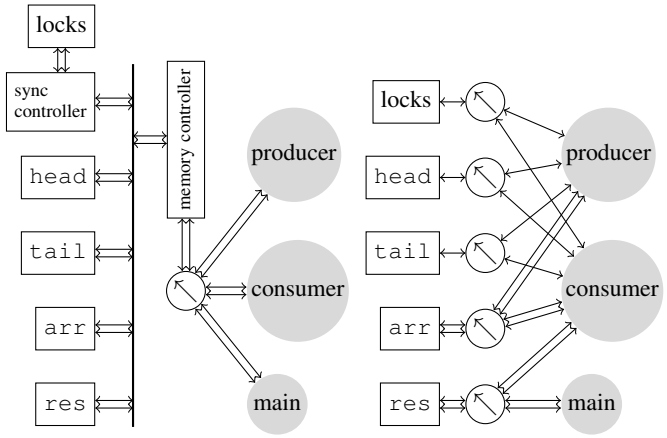
We describe all the common atomic design points for both data structures, including our method and prior work, in §V-A. We also discuss our hardware implementations via two different tools in §V-B. We conclude our evaluation by aggregating the overall resource utilisation of our method compared to prior work in §V-E.

A. Design points

We evaluate both these data structures on seven design points (as shown in Table I): two unsound versions, one lock-

[5]Our full Alloy model files are available online [11].

[6]http://en.cppreference.com/w/cpp/atomic/memory_order



(a) LegUp 4.0 (b) LegUp 5.1

Fig. 4. An example of the generated memory architecture for SPSC buffer by LegUp 4.0 and 5.1 respectively. In general, LegUp 4.0 treats all shared memory as global memory whereas LegUp 5.1 performs points-to analysis to infer variable-local arbitration [37].

```
atomic_int flag1 = 0, ..., flagN = 0;
int data1 = 0, ..., dataN = 0;
```

```
1.1  for(i=0; i<ITER; i++) {
1.2    if(ld(&flag1,ACQ)==0){
1.3      data1++;
1.4      st(&flag1,1,REL);
1.5    }
       ...
1.7    if(ld(&flagN,ACQ)==0){
1.8      dataN++;
1.9      st(&flagN,1,REL);
1.10   }
1.11 }
```

```
2.1  for(i=0; i<ITER; i++) {
2.2    if(ld(&flag1,ACQ)==1){
2.3      data1++;
2.4      st(&flag1,0,REL);
2.5    }
       ...
2.7    if(ld(&flagN,ACQ)==1){
2.8      dataN++;
2.9      st(&flagN,0,REL);
2.10   }
2.11 }
```

Fig. 5. A two-threaded message-passing example with *acquire-release* semantics on N independent channels.

based version, and four lock-free versions. The first of the two unsound versions is *Unsound*. This design uses neither atomics nor locks when implementing our buffers. *Pipelined unsound* is the *Unsound* design with loop pipelining enabled. Both of these designs can yield incorrect results, as shown in §III; we include them because they provide an experimental upper limit on performance.

We implement one lock-based version, *Atomic locks*, using pre-existing lock support provided by both HLS tools. *Atomic locks* represents the simplest way of implement atomics using pre-existing HLS tools. Atomics can be synthesised correctly by wrapping locks around each atomic access. *Atomic locks* is the finest-grained approach: every individual atomic load/store operation is wrapped in its own critical section, with one lock per data structure. This locking strategy is similar to the support that LegUp currently offers for implementing OpenMP atomics [30], except we assign one lock per data structure and avoid using a single global lock for all threads to reduce contention.

We implement all four of our MCM extensions, discussed in §IV, that support C11 atomics. *SC atomics* implements the MCM from §IV-A that implements all atomics as sequentially consistent. This design is the strictest but easiest-to-implement. *Pipelined SC atomics* implements the MCM from §IV-C that adds pipelining support to *SC atomics*. *Weak atomics* implements the MCM from §IV-B that differentiates C11 atomics by their consistency mode. This design has the best potential for performance but is harder to implement. *Pipelined weak atomics* implements the MCM from §IV-D that extends *Weak atomics* to enable loop pipelining. This MCM can further exploit the performance of weak atomics by allowing interleaving of multiple loop iterations while maintaining program correctness.

B. Hardware implementation

We implement our case study using two tools: LegUp 4.0 and LegUp 5.1. For both tools, we synthesise each Pthread as a hardware accelerator, with shared memory implemented on the FPGA. We place-and-route all the designs for a Cyclone V SoC FPGA (5CSEMA5) with 32075 ALMs, 128300 registers, and 3970 Kb of RAM blocks.

We treat LegUp 4.0 and LegUp 5.1 as two different tools because the generated memory architecture of these two versions are different. Figure 4 shows an example of the generated memory architecture by LegUp 4.0 and 5.1 for the SPSC buffer. Both tools instantiate each shared memory variable/array as individual memory elements (`head`, `tail`, `arr` and `res`). Although each variable/array is synthesised as an individual memory element, LegUp 4.0 requires all shared memory access to be arbitrated via a single global memory controller, as seen in Figure 4a. Each thread only has two-ported access to shared memory on LegUp 4.0 that enforces an added resource constraints to the intra-thread scheduling constraints. In contrast, LegUp 5.1 can analyse the access patterns of each thread via points-to analysis and generates variable-local arbitration for each shared memory element, referred to as local and shared-local memories in [37], as seen in Figure 4b. In Figure 4b, we see that each variable is now protected by an individual arbiter and threads can access all variables in parallel (one port per register and two ports per RAMs) without having to be restricted by two-ported access of shared memory in LegUp 4.0. We show in §V-C2 and §V-D5 how the different arbitration schemes impact the throughput as we increase the workload of memory accesses. In addition to different arbitration schemes, LegUp 4.0 and 5.1 also synthesise locks differently. On LegUp 4.0, locks are synthesised centrally and protected by a synchronisation controller that is accessed by the global memory controller. Instead on LegUp 5.1, locks are synthesised distributedly as individual memory elements and each thread spins on individual locks to obtain and surrender lock access.

C. Pilot experiment: Message-passing

Our first experiment is a message-passing application in which two threads take turns to pass messages to each other via shared variables. This application combines two well-known litmus tests for weak atomics: message-passing and load-buffering [6].

1) Code behaviour: Figure 5 shows two threads operating on N message-passing channels. Each thread must acquire

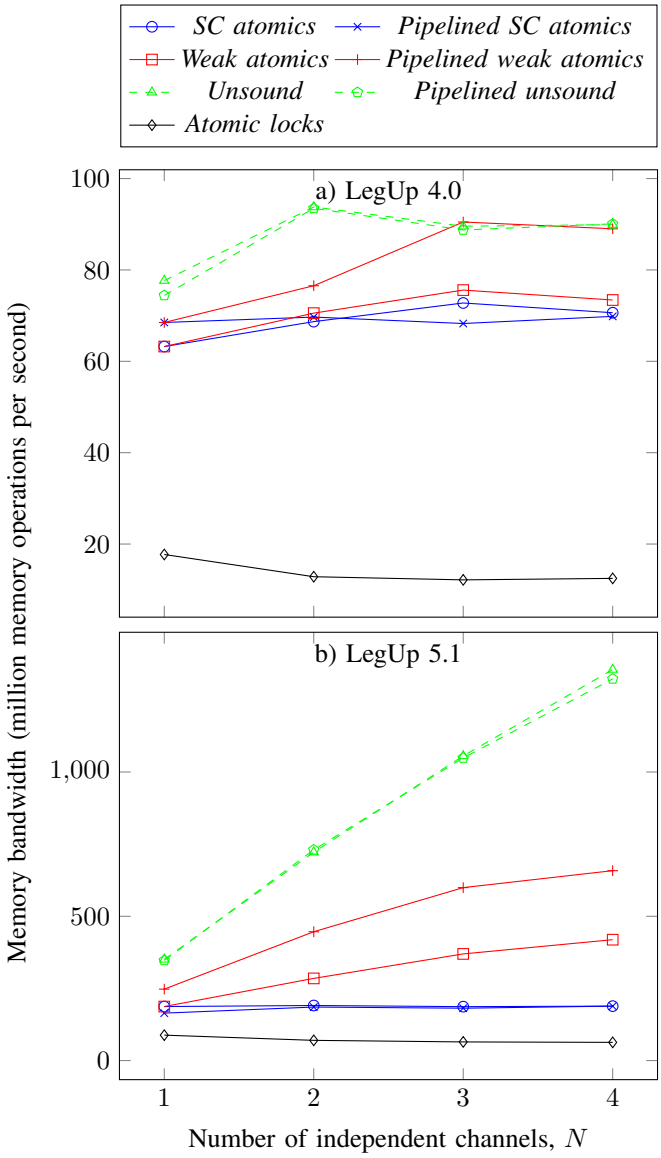


Fig. 6. Throughput results for our pilot experiment. The dashed lines indicate unsound designs; we include their throughput only as a limit study of the best performance possible.

the atomic flag (line 1.2 and line 2.2) before reading the data (line 1.3 and line 2.3), preserving load-buffering behaviour. Each thread must also write to data (line 1.3 and line 2.3) before releasing the atomic flag (line 1.4 and line 2.4), preserving message-passing behaviour. We perform message-passing across multiple channels for a fixed number of iterations to measure performance across all our design points.

2) Throughput: Figure 6 shows memory bandwidth achieved as we scale the number of independent channels for both tools across all design points. On both LegUp 4.0 and 5.1, the *Unsound* and *Pipelined unsound* designs achieve the highest bandwidth but do not guarantee correct execution. These lines are treated as limits to the maximum achievable bandwidth for our atomic implementations.

Atomic locks achieves the worst bandwidth of 14 million memory operations per second (Mmops) and 71 Mmops for LegUp 4.0 and 5.1 respectively, although we see a 5x improvement when arbitration is localised in LegUp 5.1, as discussed in §V-B. Both tools, however, restrict any improvements as we scale the number of channels.

SC atomics achieves up to 6x and 3x bandwidth improvements compared to *Atomic locks* on LegUp 4.0 and 5.1 respectively. These results show that wrapping locks around atomic memory access results in unnecessary performance overhead. *SC atomics* also restricts performance when scaling the number of channels since SC atomics serialise accesses to each channel. Each thread is required to exchange messages on one channel at a time.

Weak atomics achieves up to 6x and 6x bandwidth improvements compared to *Atomic locks* on LegUp 4.0 and 5.1 respectively. *Weak atomics* only achieves 3% improvement compared to *SC atomics* on LegUp 4.0 but achieves up to 2x improvement on LegUp 5.1. *SC atomics* and *Weak atomics* have similar performance on LegUp 4.0 because the number of memory accesses are limited to two accesses per cycle due to the resource constraints imposed by global access of shared memory, as discussed in §V-B. *Weak atomics* allows for better performance when scaling the number of channels, since weak atomics allow each thread to exchange messages across multiple channels at a time.

When loop pipelining is enabled, *Pipelined SC atomics* achieves similar bandwidth to *SC atomics* on both tools. Since the final memory access in the loop (line 1.9 and line 2.9 in Figure 5) is a SC atomic, it disallows any overlapping between consecutive iterations. This restriction leads *Pipelined SC atomics* achieving a similar schedule to *SC atomics*.

In contrast, *Pipelined weak atomics* achieves up to 7x and 10x bandwidth improvements compared to *Atomic locks* on LegUp 4.0 and LegUp 5.1. Compared to *Weak atomics*, *Pipelined weak atomics* achieves up to 1.2x and 1.6x bandwidth improvements on LegUp 4.0 and 5.1, since weak atomics permits overlapping of consecutive iterations. In particular, the release atomic in line 1.9 (of Figure 5) is allowed to overlap/reorder with the next iteration's memory operations from line 1.1 up to line 1.7.

D. Case study: Circular Buffer

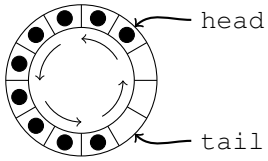
Thus far, our code examples have been relatively small, and designed to convey the problems of weak behaviour and demonstrate the potential of strengthening MCMs to implement atomics. We investigate the performance of SC atomics and weak atomics on a real-world example: a lock-free single-producer-single-consumer circular buffer, due to Hedström [38], as our case study. 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 [39].

1) Code behaviour: Figure 7 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 Fig. 8.

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

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

Fig. 7. A single-producer-single-consumer circular buffer that synchronises the producer (left) and the consumer (right) using weak C11 atomics [38].



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

Fig. 8. The SPSC circular buffer, diagrammatically. This buffer consists of an array of elements protected by `head` and `tail` pointers, each pointing to the next location to pop from and push to respectively.

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 Fig. 8. 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.

Hedström explains in detail why each memory access does not require full SC [38]. 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.

2) Ensuring correctness: Going beyond the informal argument presented above, how can we be confident that the program we have chosen for evaluating our implementations of weak atomics actually uses weak atomics correctly? Ensuring the correctness of any multi-threaded program in a weakly consistent setting is difficult because of the counterintuitive behaviours allowed by a weak MCM. Ordinary testing is inconclusive because of the non-deterministic nature of multi-threading and 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 [6] to

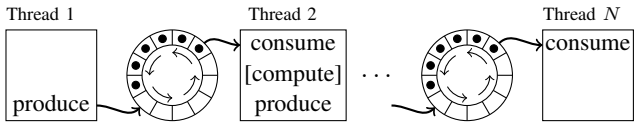


Fig. 9. Our experiments (for N from 2 to 9). For CHAINING, [compute] is a no-op; for DIVISION, it is a long-latency division operation.

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 operations, we remove the while-loops. We give the actual code we verified in our companion material [11]. 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.

3) Experiment Setup: We conduct experiments based on the circular buffer by chaining together several of these buffers and observing the performance and area characteristics of our various designs. Figure 9 shows our basic experimental setup: a producer thread sends a stream of messages across a chain of repeater threads to a consumer thread that verifies the results. We conduct two experiments based on this setup. The first is CHAINING, where we solely observe the maximum performance on the memory operations, and the second is DIVISION, where we introduce a long-latency divide operation into each repeater thread (between the 'consume' and 'produce' steps) to analyse the impact of computation on our designs. In both versions, we observe the performance and scalability of our various designs and HLS tools by increasing the number of repeater threads.

4) Additional lock-based design points: In addition to the design points discussed in §V-A, we further optimise *Atomic locks* for better performance by reducing the number of locking and unlocking steps for the SPSC circular buffer. The intention for this exercise is to observe the performance of the buffer when coarsening the locking granularity. As we coarsen the granularity of the implementations, the critical sections grow in size and move towards lock-based programming rather than exploiting the buffer's lock-freedom. In comparison to *Atomic locks*, *Fine locks* also uses one lock per buffer but requires a repeater thread only to hold the lock of its source buffer while consuming, and then only to hold the lock of its target buffer while producing. This is slightly coarser-grained locking strategy than *Atomic locks* and is intended to allow any computation by the repeater thread, between the 'consume' and 'produce' steps, to avoid being in a critical section. *Coarse locks* is the coarsest approach and requires each repeater thread to hold the lock of both its source and its target buffer before it begins consuming or producing. This design only requires a pair of locking and unlocking steps.

5) Throughput: Figure 10 shows the performance of CHAINING and DIVISION for both HLS tools, measured in terms of the number of data packets transferred per second.

The two unsound designs give the highest throughput, but because they can give incorrect results, we include them only

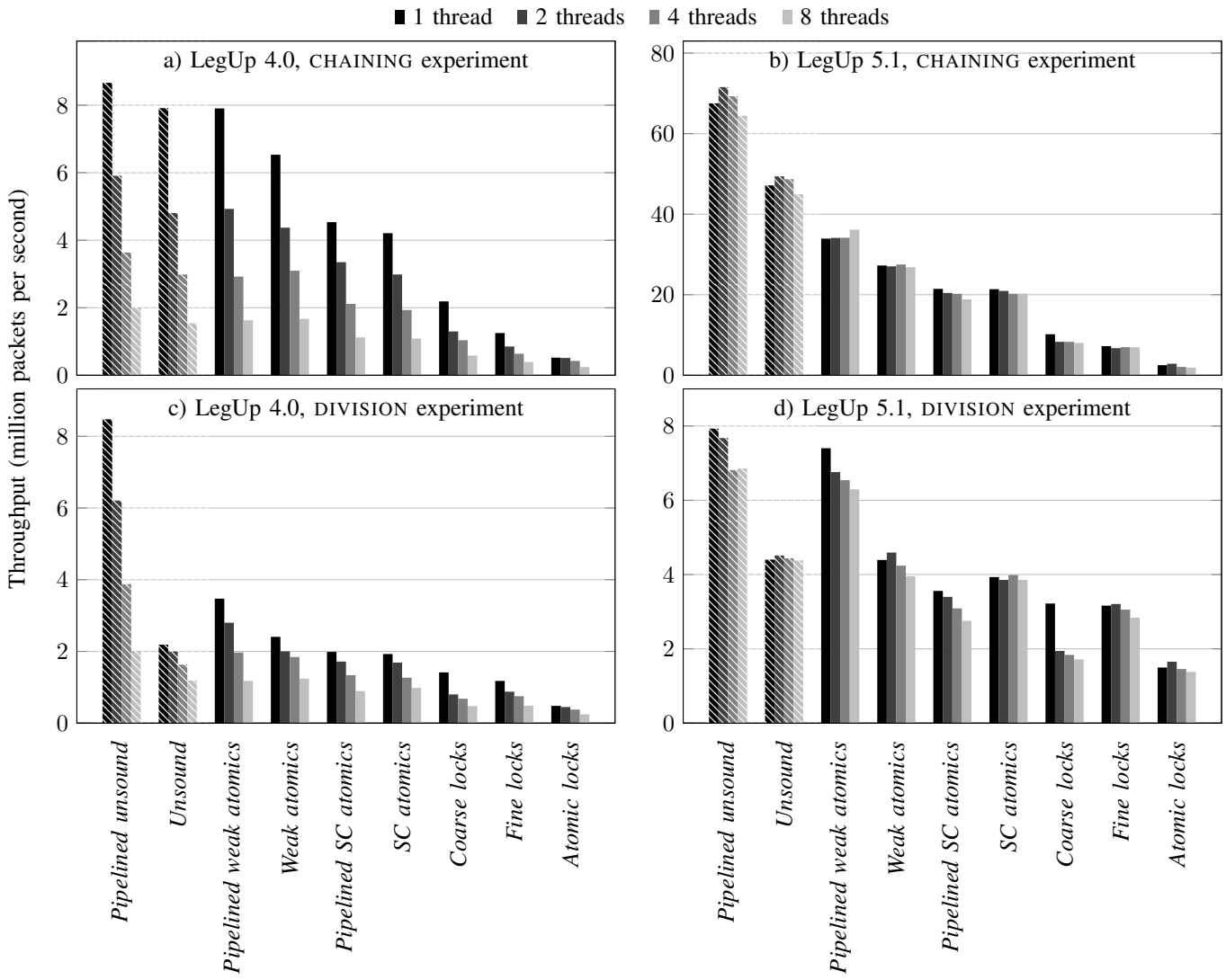


Fig. 10. Throughput results. The hatched bars indicate unsound designs; we include their throughput only as a limit study of the best performance possible.

as nominal upper bounds on performance. For instance, in the *Pipelined unsound* case, we were occasionally able to observe packets being erroneously duplicated.

The lock-based designs are the slowest in both experiments. Although the results of these designs are correct, correctness comes with a high overhead. On average, the lock-based designs are 6x slower than *Unsound*. LegUp 4.0's locks are 5x slower than those in LegUp 5.1's, due to the improvement on lock access on LegUp 5.1 as discussed in §V-B. On average, *Coarse locks* is 1.1x faster than *Fine locks*, potentially because it involves fewer lock and unlock steps. However, *Fine locks* achieves 1.5x better performance than *Coarse locks* for DIVISION, which can be attributed to *Fine locks* not holding any locks during the division operation (critical section). *Atomic locks* is consistently the least-performing lock-based implementation and performs 12x slower than *Unsound*, on average.

SC atomics performs on average 8.2x faster than *Atomic locks* in the CHAINING experiment (and 3x faster in DIVISION). *SC atomics* restricts the ordering of the atomic operations because of the additional memory dependencies but does not incur the overheads of acquiring and releasing locks. The benefit of *SC atomics* reduces during the DIVISION experiment when the compute latency dominates the total latency of the repeater's schedule. *SC atomics* can recover 87% of the *Unsound* throughput for DIVISION (which is compute-bound) but only 44% for CHAINING (which is memory-bound). *SC atomics* also performs 6x better on LegUp 5.1 than on LegUp 4.0, due to the improved memory architecture discussed in §V-B.

Weak atomics is 1.3x and 1.2x faster than *SC atomics* for CHAINING and DIVISION. This shows that although weak atomics are harder to implement, they offer superior performance in our case study. On average, *Weak atomics* achieves reduced repeater latency compared to *SC atomics* because weak atomics require fewer dependencies than SC atomics. *Weak atomics* performs on average 11x and 3.4x faster than *Atomic locks* in the CHAINING and DIVISION experiments. *Weak atomics* recovers 59% of the *Unsound* throughput for CHAINING, and actually outperforms *Unsound* by 0.3% for

DIVISION. This is because despite *Weak atomics* having a slightly higher cycle count, it runs at a much higher clock frequency than *Unsound*. *Weak atomics* also performs 5x better on LegUp 5.1 than LegUp 4.0, once again due to the improved memory architecture discussed in §V-B.

We further extend both our atomic MCMs to support loop pipelining. *Pipelined SC atomics* represents the pipelined version of *SC atomics*. Both designs achieve a similar performance (pipelining worsens performance by 2.8%). Since *Pipelined SC atomics* treats all atomics as sequentially-consistent atomics, and because the last memory operation in the loop is an atomic store, there is little opportunity for pipelining the repeater. *Pipelined SC atomics* recovers only 44% and 37% of *Pipelined unsound* throughput for CHAINING and DIVISION.

Pipelined weak atomics is the best-performing (correct) design across all our experiments. In contrast to *Pipelined SC atomics*, *Pipelined weak atomics* performs 1.25x and 1.45x better in CHAINING and DIVISION than its non-pipelined counterpart *Weak atomics*. In our case study, weak atomics allow more overlapping between memory operations in consecutive iterations compared to SC atomics. Since the last memory operation of the loop is a release atomic store, memory operations from the next iteration are allowed to overlap to some degree. In fact, overlapping is allowed up to the acquire atomic load in Line 1.2 of Fig. 7 of the current iteration. In our experiments, this design's initiation interval is roughly half its loop latency. *Pipelined weak atomics* recovers on average 53% and 72% of the *Pipelined unsound* implementation for CHAINING and DIVISION. In contrast to all other designs, this design offers better performance for DIVISION, as its effect on performance is much greater for large latencies (the initiation interval being half the loop latency). Overall, *Pipelined weak atomics* achieves an average 14x and 5x speedup in the CHAINING and DIVISION experiments, compared to *Atomic locks*.

E. Overall resource utilisation

Figure 11 shows the resource utilisation of our atomic implementations relative to *Atomic locks*. On average, *SC atomics* and *Weak atomics* only use 90% and 93% of the LUTs required by *Atomic locks* despite having an average of 6x and 7.8x throughput improvement for our case study. An additional 7% and 4% LUT usage is required to implement *Pipelined SC atomics* and *Pipelined weak atomics*, to achieve 1x and 1.3x compared to *SC atomics* and *Weak atomics* respectively. Loop pipelining introduces additional pipeline stalling logic and validity signals that increases LUT usage. Despite added circuitry to support loop pipelining, both these designs still use fewer LUTs than *Atomic locks*, on average. *Pipelined weak atomics* only uses 15% additional LUTs to achieve an average throughput performance of 10x. Overall, our atomic implementations use between 79% and 115% LUT compared to *Atomic locks*.

Our atomic implementations also required fewer registers on average compared *Atomic locks*. Pipelining does not introduce additional register usage as the pipelining circuitry is mostly combinational. Our atomic implementations use between 80% and 122% register usage compared to *Atomic locks* for average performance improvement of 10x.

VI. Conclusion

We have investigated how multi-threaded C programs that use atomic operations to synchronise threads can be synthesised to FPGAs by extending two HLS tools (LegUp 4.0 and LegUp 5.1). Where previous approaches have relied on locks to implement atomics, we have shown how they can instead be implemented by injecting additional intra-thread constraints during the HLS scheduling phase. The C standard defines a range of atomic operations, some of which are sequentially consistent, others weakly consistent, and we have defined scheduling constraints that are sensitive to the consistency mode of each operation. Atomics support loop pipelining, unlike lock-based solutions, which give rise to critical sections that cannot overlap. By adding further scheduling constraints between loop iterations, we have shown how to add support for pipelining.

To justify the correctness of our work, we have used an automatic model checker to prove that all programs (up to a bounded size) will be synthesised correctly.

We have evaluated the performance of our work on a widely-used lock-free buffer from the Linux kernel and the Boost library. We demonstrate a 6x speedup compared to LegUp's current lock-based implementation of atomics [30]. We further demonstrate that switching from sequentially consistent atomics to weakly consistent atomics (where safe to do so) yields a further 1.3x speedup, and that enabling pipelining brings a further 1.3x speedup. We achieve an overall performance of 10x compared LegUp's lock-based implementation of atomics with fewer LUT and registers on average and with a worst-case 20% increase in LUT and register overheads.

Where previous work on implementing weak atomics has concentrated on mapping C to processor-specific assembly code [6], [5], our work shows how HLS can compile the C standard for 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 Jason Anderson, Mark Batty, Shane Fleming, and David Thomas 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, an Imperial College Research Fellowship (Wickerson), and a Royal Academy of Engineering / Imagination Technologies Research Chair (Constantinides) is gratefully acknowledged.

References

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

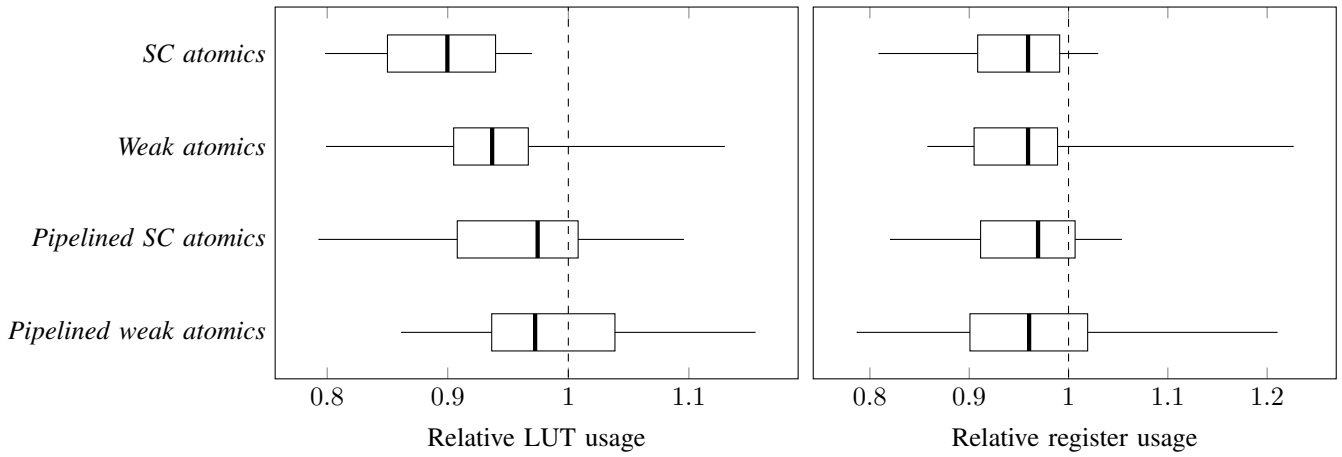


Fig. 11. Resource utilisation, all relative to the *Atomic locks* implementation. Each line depicts the minimal, maximal, first and third quartiles and median utilisation, which are aggregated over all experiments and both HLS tools.

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

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

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

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

[6] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing C++ concurrency," in *Principles of Programming Languages (POPL)*, 2011.

[7] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "GPU concurrency: weak behaviours and programming assumptions," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[8] D. Jackson, *Software Abstractions – Logic, Language, and Analysis*, revised edition ed. MIT Press, 2012.

[9] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Field-Programmable Gate Arrays (FPGA)*, 2011.

[10] N. Ramanathan, S. T. Fleming, J. Wickerson, and G. A. Constantinides, "Hardware Synthesis of Weakly Consistent C Concurrency," in *Field-Programmable Gate Arrays (FPGA)*, 2017.

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

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

[13] Xilinx, *Vivado Design Suite User Guide: High-Level Synthesis (v2016.2)*, 2016.

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

[15] Xilinx, *SDAccel Development Environment - User Guide (v2016.2)*, 2016.

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

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

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

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

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

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

[22] M. D. Sinclair, J. Alsop, and S. V. Adve, "Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems," in *Proceedings of the 44th Annual International Symposium on Computer Architecture, (ISCA)*. ACM, 2017.

[23] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, S. Brown, and J. Anderson, "The effect of compiler optimizations on high-level synthesis for FPGAs," in *Field-Programmable Custom Computing Machines (FCCM)*, 2013.

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

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

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

[27] B. R. Rau and C. D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing," in *14th Annual Workshop on Microprogramming, (MICRO)*. IEEE Press, 1981.

[28] Z. Zhang and B. Liu, "SDC-based modulo scheduling for pipeline synthesis," in *International Conference on Computer-Aided Design*. IEEE Press, 2013.

[29] A. Canis, S. D. Brown, and J. H. Anderson, "Modulo SDC scheduling with recurrence minimization in high-level synthesis," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. IEEE, 2014.

[30] 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.

[31] M. Dodds, M. Batty, and A. Gotsman, "Compositional verification of relaxed-memory program transformations," 2016, under review. [Online]. Available: bit.ly/2bmY7wI

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

[33] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W.-m. W. Hwu, "Characterizing the impact of predicated execution on branch prediction," in *Microarchitecture (MICRO)*, 1994.

[34] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides, "Automatically comparing memory consistency models," in *Principles of Programming Languages (POPL)*, 2017.

[35] 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.

[36] J. Wickerson, "cppreference.com gets acquire/release instructions wrong," 2016. [Online]. Available: http://bit.ly/2qOaTX9

[37] J. Choi, S. Brown, and J. Anderson, "Resource and memory management techniques for the high-level synthesis of software threads into parallel FPGA hardware," in *Field-Programmable Technology (FPT)*, 2015.
[38] K. Hedström, "Lock-free single-producer-single-consumer circular queue," 2014, bit.ly/2dbr8IK.
[39] T. Blechmann, "Lock-free single-producer/single-consumer ringbuffer," 2013, bit.ly/2dbqFq1.

Nadesh Ramanathan Nadesh Ramanathan received an MPhil in Advanced Computer Science from the University of Cambridge in 2014. He is currently pursuing his PhD in the Department of Electrical and Electronic Engineering at Imperial College London, as a member of the Circuits and Systems research group. His current research interest is on high-level synthesis for multithreaded programs.

John Wickerson John Wickerson (M17) received a PhD in Computer Science from the University of Cambridge in 2013. He currently holds an Imperial College Research Fellowship in the Department of Electrical and Electronic Engineering at Imperial College London. His research interests include high-level synthesis, the semantics of programming languages, and software verification.

George A. Constantinides George A. Constantinides (S96-M01-SM08) received the Ph.D. degree from Imperial College London in 2001. Since 2002, he has been with the faculty at Imperial College London, where he is currently Royal Academy of Engineering / Imagination Technologies Research Chair, Professor of Digital Computation, and Head of the Circuits and Systems research group. He has served as chair of the FPGA, FPL and FPT conferences. He currently serves on several program committees and has published over 150 research papers in peer refereed journals and international conferences. Prof Constantinides is a Senior Member of the IEEE and a Fellow of the British Computer Society.