Concurrency-Aware Thread Scheduling for High-Level Synthesis

Nadesh Ramanathan, George A. Constantinides and John Wickerson
Department of Electrical and Electronic Engineering
Imperial College London, London, SW7 2AZ, United Kingdom
E-mail: {n.ramanathan14, g.constantinides, j.wickerson}@imperial.ac.uk

***Abstract*—When mapping C programs to hardware, high-level synthesis (HLS) tools seek to reorder instructions so they can be packed into as few clock cycles as possible. However, when synthesising multi-threaded C, instruction reordering is inhibited by the presence of *atomic operations* ('atomics'), such as compare-and-swap. Atomics, the fundamental concurrency primitive in C, are the basis of more abstract concurrency mechanisms such as locks, and also of efficient lock-free data structures.**

Whether a particular atomic can be legally reordered within a thread can depend on the memory access patterns of other threads. Existing HLS tools that support atomics typically schedule each thread independently, and so must be conservative when optimising around atomics. Yet HLS tools are distinguished from conventional compilers by having the entire program available. Can this information be exploited to allow more reorderings within each thread, and hence to obtain more efficient schedules?

In this work, we propose a global analysis that determines, for each thread, which pairs of instructions must not be reordered. Our analysis is sensitive to the C consistency mode of the atomics involved (e.g. relaxed, release, acquire, and sequentially-consistent). We have used the Alloy model checker to validate our analysis against the C language standard, and have implemented it in the LegUp HLS tool. An evaluation on several lock-free data structure benchmarks indicates that our analysis leads to a 1.6× average global speedup.

I. Introduction

When mapping C programs to hardware, high-level synthesis (HLS) tools try to schedule instructions into as few clock cycles as possible. To do this, they seek opportunities to reorder or parallelise instructions without affecting the program's behaviour. However, when synthesising multi-threaded C programs, changes to the order of instructions are inhibited by the presence of *atomic* operations.

Atomic operations are the fundamental concurrency primitive provided by the C language [2, §7.17]. They can be used to implement more programmer-friendly concurrency mechanisms such as mutual exclusion locks, and also to build efficient lock-free concurrent data structures. Atomics are guaranteed to appear to execute instantaneously. They also impose constraints on the ordering of memory accesses.

In our previous work, we showed that atomics can be implemented in HLS by adding scheduling constraints within each thread to ensure correct global ordering [3]. These constraints were determined locally; i.e., based only on *one* thread's memory accesses. In the current work, we exploit the fact that HLS tools have the entire program available: we determine each thread's scheduling constraints by analysing the memory accesses of *all other threads*. This can lead to a significant reduction in the number of constraints required, as demonstrated in the following example.

Example 1: Consider the following program, which consists of an ordinary (non-atomic) store to `x` followed by an atomic store to `y`:

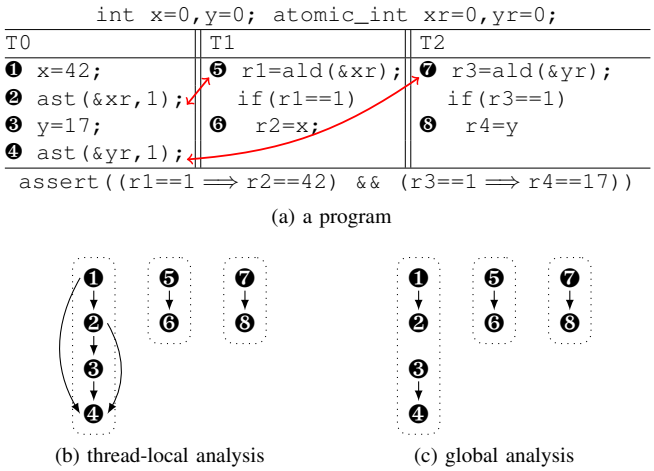
```
x=1;
atomic_store(&y,1);
```
(Program 1)

Assuming `x` and `y` do not alias, these two instructions can safely be parallelised. The analysis proposed in this paper correctly determines that both can be scheduled into the same clock cycle. Unfortunately, compilers and HLS tools that use only a thread-local analysis have to work on the assumption that there might be other threads concurrently accessing `x` and `y`, so they cannot safely reorder the two stores. For instance, there could be an additional thread that loads atomically from `y` and then non-atomically from `x`, as shown below, where || separates the two threads:

```
x=1;                    ||  if(atomic_load(&y))
atomic_store(&y,1);     ||    r0=x;
```
(Program 2)

The two stores must no longer be reordered. This is because the C standard dictates that when an atomic load observes an atomic store in another thread, the two threads synchronise [2, §5.1.2.4.11]. As a consequence, all memory accesses that happen before the atomic store are guaranteed to become visible to all memory accesses after the atomic load. This guarantee could be violated in Program 2 if the stores are executed out of order.

In this paper, we propose a global analysis that determines which pairs of instructions must not be reordered within a thread. Our analysis handles programs that use *sequentially-consistent* (SC) atomics (the default consistency mode), as well as 'weak' atomics [2, §7.17.3]. Weak atomics impose fewer ordering constraints than SC atomics, and hence have more opportunities for reordering and parallelism, but their complex semantics makes them harder for programmers to use correctly. Our analysis handles the *relaxed*, *acquire*, *release*, and *acquire-release* weak consistency modes. We use the Alloy model checker to validate our analysis against the C standard for both SC and weak atomics.



(a) a program

(b) thread-local analysis

(c) global analysis

Fig. 1. Three-threaded message passing example with two channels.

We implement our analysis as an LLVM pass in the LegUp 5.1 HLS tool [4], as detailed in §III. To evaluate it (§IV), we compile three real-world lock-free data structures: the Treiber stack [5], a single-producer-single-consumer buffer [6] and the Michael–Scott queue [7]. Our results show that our analysis leads to an average 1.6× global speedup compared to our previous thread-local analysis, with per-benchmark speedups ranging up to 3.7×. Making the analysis sensitive to weak atomics leads to an average 1.2× hardware speedup compared to treating all atomics as SC.

Our companion material [1] includes our benchmarks, Alloy model files, and performance data.

II. Motivating Example

In this section, we give a more realistic program that can benefit from our whole-program analysis, and explain intuitively how the analysis works.

Consider the program in Fig 1a, in which `ald` stands for `atomic_load` and `ast` stands for `atomic_store`. The program uses atomic variables to pass messages from thread `T0` to threads `T1` and `T2`. This is an important concurrent programming pattern, as it represents a master thread distributing work to two other threads. Thread `T0` passes messages to `T1` and `T2` by first writing to a non-atomic variable (`x` or `y`) and then writing to an atomic variable that serves as a ready signal (`xr` or `yr`). Threads `T1` and `T2` receive `T0`'s messages by checking that their ready signals are set before reading the data. The `assert` ensures that there are no message-passing violations; i.e., that `T1` and `T2` receive 42 and 17 respectively if their ready signals are set.

The arrows in Fig. 1b show the ordering constraints that a thread-local analysis would impose [3]. These constraints are injected on the basis that ❷ and ❹ are atomic, and hence they must not be reordered with other memory accesses. This forces all four memory operations to be serialised during scheduling.

However, considering the program as a whole, the ordering constraints in `T0` are conservative, because we do not need to enforce ordering between the two independent message-passing channels. Our analysis considers which pairs of atomics could actually synchronise at runtime. As shown by the arrows in Fig. 1a, `T0` can synchronise with `T1` if ❺ observes ❷ and with `T2` if ❼ observes ❹. If these operations do synchronise, then we must ensure that all memory accesses before the atomic store are visible to all memory accesses after the atomic load. We can ensure this by adding two ordering constraints in `T0` and one constraint each in `T1` and `T2`, as shown in Fig 1c. All other accesses can be safely reordered because no thread would be able to observe such a reordering. Hence, our global analysis reduces the number of ordering constraints in `T0` from five to two, and allows ❸ and ❹ to be scheduled in parallel with ❶ and ❷.

In summary, analysing the possible synchronisation opportunities of the entire program can allow us to reduce the ordering constraints within each thread.

III. Method

We implement our analysis in LegUp 5.1 as an LLVM module pass between the allocation stage and the scheduling stage. From the LLVM intermediate representation (IR) we extract all memory operations of all threads along with their locations and consistency modes. We also verify that memory locations do not alias using LLVM's alias analysis tool.

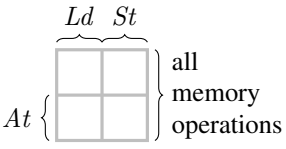
Using this information, we perform our global analysis, which identifies the ordering constraints to be preserved. We subsequently inject these constraints into LegUp's scheduler. Our analysis supports atomic loads, stores, and compare-and-swaps, both on scalar variables and on arrays. We also support fences, but do not discuss them in this paper to simplify the presentation. Our analysis assumes that reads and writes to FPGA memory elements occur instantaneously (i.e., no caches or write buffers) – that is, the only source of reordering of memory accesses is the instruction scheduler.

LegUp divides a thread into several basic blocks of straight-line code. Instructions within a basic block can be reordered and parallelised, but basic blocks are executed in sequence. Hence, our analysis only needs to produce ordering constraints within a basic block.

A. Input to our Analysis

The input to our analysis is a set of memory operations grouped into the following overlapping subsets:

- Ld, the set of loads,
- St, the set of stores, and
- At, the set of all atomics.



Our analysis also relies on the following relations between those operations:

- po, the 'program order' relation, which relates all the memory accesses within each thread in a strict total order, as stipulated by the programmer,
- $sloc$, the 'same location' relation, which relates all accesses to the same memory location (as determined by an alias analysis), and
- $sthd$, the 'same thread' relation, which relates all accesses within the same thread.

B. Identifying instructions that must not be reordered

Our analysis begins by identifying pairs of operations that can cause threads to synchronise:

$$canSync = (At \times At) \cap sloc \setminus sthd.$$

The *canSync* relation connects any two atomic operations on the same location from different threads. For instance, the *canSync* edges of our motivating example are given by the arrows in Fig. 1a. If two operations in *canSync*, say A and B, do synchronise at runtime, then all memory operations that A has observed must become visible to operations that follow B. For instance, if ❷ synchronises with ❺ then operation ❶ must be visible to ❻. In order to ensure this, both *po* edges (❶,❷) and (❺,❻) must be preserved.

In general, we must consider not just isolated *canSync* edges, but *paths* of them, in order to handle programs like the one shown below.

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

T0	T1	T2
x=17;	r1=ald(&y);	r2=ald(&z);
ast(&y,1);	if(r1==1)	if(r2==1)
	ast(&z,1);	r3=x;

```
assert((r1==1 ∧ r2==1) ⟹ r3==17)
```

Here, thread `T0` can synchronise with `T2` indirectly, via thread `T1`, as shown by the arrows. If both flags `y` and `z` are observed, then `T2` must observe the value of `x` that is written by `T0`, as captured by the assertion. This program shows that the global order in which memory accesses are allowed to occur can depend on paths between the accesses that are made up of several *canSync* edges.

Hence, to enumerate all possible synchronisation opportunities, we construct the following set of paths through the memory operations of the entire program. A path is an ordered list of edges, and each edge is a pair of *po*-related operations. The set *AllPaths* is defined to contain the path $[(v_0, v_0'), \ldots, (v_n, v_n')]$ if and only if it satisfies all of the following conditions:

$$\forall i.\, 0 \le i \le n \implies (v_i, v_i') \in po \quad (1)$$

$$\forall i.\, 0 \le i < n \implies (v_i', v_{i+1}) \in canSync \quad (2)$$

$$\forall i, j.\, 0 \le i < j \le n \implies (v_i, v_j) \notin sthd \quad (3)$$

$$(v_0, v_n') \in sloc \quad (4)$$

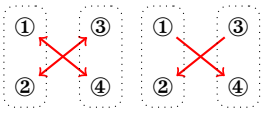
$$(v_0, v_n') \in Ld \times Ld \implies (v_0, v_n') \in At \times At \quad (5)$$

Condition (1) states that every path is a ordered list of $n+1$ edges from *po*. Condition (2) states that the target operation of each *po* edge is connected to the source operation of the next *po* edge in the path via *canSync*. We only consider paths that do not revisit a thread (Condition 3), since such paths can be minimised by removing the detour. This condition limits n to be smaller than the number of threads. Also, we are only interested in paths that start and end with accesses to the same location (Condition 4) because the order in which memory accesses to different locations occur cannot be directly observed. Finally, if a path begins and ends with loads, then we only consider it if both loads are atomic (Condition 5). Non-atomic loads can be reordered because this reordering cannot be observed unless the program has a data race [8], and this would be a programming error.

```
atomic_int x=0, y=0;
```

① ast(&x,1, release);	③ ast(&y,1, release);
② r1=ald(&y, acquire);	④ r2=ald(&x, acquire);

(a) a program



(b) SC (c) Weak

Fig. 2. The 'store buffering' programming pattern, and its *canSync* edges under SC and weak consistency

As an example, there are two paths in Fig 1a that satisfy Conditions (1) to (5): $[(❶,❷),(❺,❻)]$ and $[(❸,❹),(❼,❽)]$.

Ultimately, having enumerated all relevant paths, we preserve all the *po* edges that appear in at least one path. That is, we define the preserved program order, *ppo*, as follows:

$$ppo = \{(v, v') \mid \exists p \in AllPaths.\, (v, v') \in p\}. \quad (6)$$

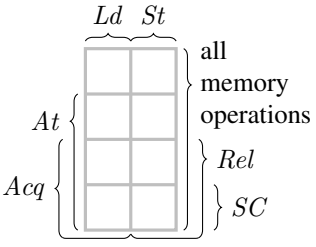
The only edges that need to be preserved in Fig. 1a are the four *po* edges in its two paths: (❶,❷), (❺,❻), (❸,❹), and (❼,❽), as shown in Fig. 1c.

C. Exploiting weak concurrency

Thus far, we have handled all atomics using the default, strictest consistency mode: sequential consistency (SC). However, the C standard also supports a range of 'weak' atomics that offer *acquire*, *release*, and *relaxed* consistency [2, §7.17.3.1]. Weak atomics allow more reordering of memory accesses within a thread, which can improve performance, though they can be harder for programmers to use correctly. In this subsection, we describe how we extend our analysis to exploit weak atomics, then in §IV-B1, we show that the resultant reduction in scheduling constraints does indeed yield better-performing hardware.

In order to support weak atomics, we provide three further inputs to our analysis:

- *SC*, the set of SC atomics,
- *Acq*, the set of atomic loads with at least acquire consistency, and
- *Rel*, the set of atomic stores with at least release consistency.



We redefine the pairs of atomics that can cause threads to synchronise, as follows:

$$canSync = ((Rel \times Acq) \cup (SC \times At) \cup (At \times SC)) \cap sloc \setminus sthd.$$

This new definition relates two atomics to the same location on different threads if: (a) the first operation is a release atomic and the second is an acquire, or (b) either of the operations is an SC atomic. Condition (a) captures the one-way nature of release/acquire synchronisation [2, §5.1.2.4.11], while Condition (b) lets SC atomics retain their full synchronising abilities.

Figure 2a gives an example of a program that is affected by this weakening of the *canSync* relation. It illustrates the 'store buffering' pattern, which appears in, for instance, Dekker's algorithm for mutual exclusion [9]. It consists of two atomic locations, `x` and `y`, two release stores, and two acquire loads. If all of the memory accesses were SC, the outcome `r1` $=$ `r2` $=$ 0 would be forbidden by C. To ensure that this outcome cannot happen, our analysis would place *canSync* edges as shown in Fig. 2b; this would lead to paths such as $[(①, ②), (③, ④)]$ and hence both *po* edges being preserved.

However, our refined definition of *canSync* is sensitive to the program's use of release/acquire atomics. It produces one-way *canSync* edges, as seen in Fig. 2c. No legal paths can be constructed with these edges, and hence the two instructions in each thread can be reordered.

D. Compare-and-swap support

We now explain how our analysis can be extended to support atomic compare-and-swap (CAS) operations, which are central to many fine-grained concurrent algorithms [10]. A CAS operation is parameterised by an atomic location, an expected value, and a desired value. If the location holds the expected value, it is instantaneously swapped to the desired value, otherwise its value is unchanged. C defines 'strong' and 'weak' CAS operations; the difference being that a weak CAS may fail to swap even if the location *does* hold the expected value [2, §7.17.7.4.4].[1] We implement the strong CAS, as it is more powerful, and is required by our benchmarks.

1) Analysing CAS operations: For our analysis, we treat a CAS as a pair of accesses: an atomic load followed by an atomic store. The consistency modes of the load and the store are determined from the consistency mode of the original CAS. If our analysis produces a *ppo* edge that constrains either the load or the store component, it is automatically mapped to constrain the original CAS.

2) Implementing CAS operations: A straightforward method of implementing CAS operations in HLS is to perform an ordinary load and store while holding a mutual-exclusion lock [11]. However, using locks to implement atomics is inefficient because extra cycles are needed to acquire and then release the lock. Locks also prevent a CAS operation from being reordered, even if it is *relaxed*. Instead of using locks, we modify LegUp's RTL generator to support CAS operations directly in hardware.

Figure 3 shows the generated memory architecture when two threads access a shared array. The basic mechanism for accessing shared memory in LegUp is as follows. A thread asserts its enable (`en`) signal to request (`req`) access from the arbiter. On each cycle, the arbiter grants (`grant`) access only to one thread while other unsuccessful threads must stall (`stall`) and keep their enable signal asserted. To perform a CAS on a RAM, a thread requires two consecutive cycles to complete an uninterrupted sequence of read and write accesses. To achieve this effect, we first add circuitry that holds (`hold`) the arbiter's grant signal for an extra cycle, as shown in the shaded region of Fig. 3. Then, we modify each thread's state machine to pack the read and write into consecutive cycles. Finally, we implement the comparison logic between the read and write. To perform a CAS on a register, the `hold` signal is not required as registers have zero-latency reads and hence the CAS can be packed into a single cycle.

[1]Note that this usage of 'weak' is distinct from weak consistency. The strength of a CAS refers to its behaviour when the comparison succeeds; its consistency mode refers to how the CAS operation can be reordered with the other memory operations in its thread.

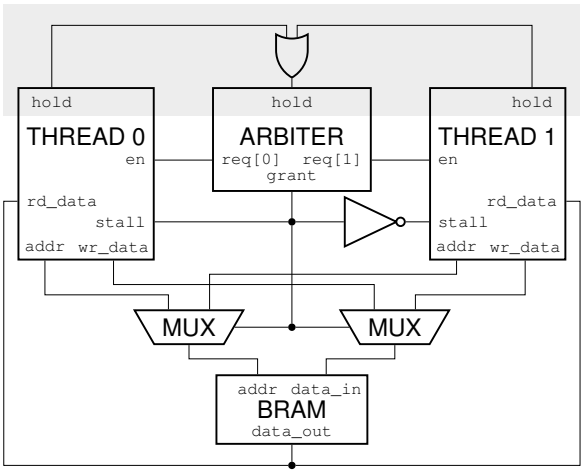


Fig. 3. A (simplified) circuit showing how CAS works for two threads, the shaded region indicating circuitry added by us.

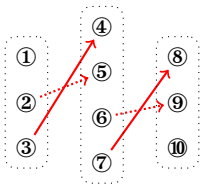


Fig. 4. Illustrating primary (solid) and secondary (dotted) *canSync* edges

E. An optimised implementation of our analysis

A naïve implementation of our analysis is to enumerate the *AllPaths* set and then to extract the *ppo* edges, as described in §III-B. However, as we show in §IV-C, this method scales poorly on realistic programs. The problem is that realistic programs have a large number of *canSync* edges and these edges can exponentially increase the number of paths to explore. To improve the scalability of our analysis, we now describe a more efficient method to calculate *ppo*. The idea is to identify a subset of the *canSync* edges as 'secondary', and to remove them while enumerating paths; then to re-introduce them on a per-path basis when calculating *ppo*.

For example, Fig. 4 shows the shape of a program with three threads and four *canSync* edges. We shall call the two dotted *canSync* edges secondary edges, because for any path that passes through one or more of these secondary edges, there always exists a path between the same endpoints that does not pass through any secondary edges. For instance, from the path $[(①, ②), (⑤, ⑥)]$ we can obtain the path $[(①, ③), (④, ⑥)]$ which passes only through 'primary' *canSync* edges.

More formally, we define the primary *canSync* edges as:

$$\begin{aligned} &\mathit{canSyncPrimary} = \\ &\quad \{(v_\mathrm{a}, v_\mathrm{b}) \in \mathit{canSync} \mid \nexists (v_\mathrm{c}, v_\mathrm{d}) \in \mathit{canSync}. \\ &\qquad (v_\mathrm{a}, v_\mathrm{c}) \in \mathit{po}^* \wedge (v_\mathrm{d}, v_\mathrm{b}) \in \mathit{po}^* \wedge \\ &\qquad (v_\mathrm{c} \neq v_\mathrm{a} \vee v_\mathrm{d} \neq v_\mathrm{b})\}. \end{aligned}$$

That is, $(v_\mathrm{a}, v_\mathrm{b})$ is a primary edge providing there exists no other *canSync* edge $(v_\mathrm{c}, v_\mathrm{d})$ such that v_c is either equal to v_a or *po*-after it, and v_d is either equal to v_b or *po*-before it. (Note that r^* is the reflexive transitive closure of r.)

We then define the set of *primary* paths, *PrimaryPaths*, as those that pass only through primary *canSync* edges, by redefining (2) to:

$$\forall i.\, 0 \leq i < n \implies (v'_i, v_{i+1}) \in \mathit{canSyncPrimary}. \qquad \text{(2a)}$$

Having calculated the set of primary paths, it remains to generate the *ppo* relation in a way that re-includes the non-primary paths. This can be done on an efficient per-path basis. The idea is, for each edge in each path, to put into *ppo* not just that *po* edge, but also any other *po* edge that a non-primary path between the same threads could have taken.

For example, in Fig. 4, the *po* edge from ④ to ⑦ is on a primary path from ① to ⑩. We include that edge in *ppo*, but also (④, ⑥), (⑤, ⑦), and (⑤, ⑥).

More formally, we generate *ppo* as follows:

$$\begin{aligned} &\mathit{ppo} = \\ &\quad \{(w_1, w_2) \mid \exists [(v_0, v'_0), \ldots, (v_n, v'_n)] \in \mathit{PrimaryPaths}. \\ &\qquad \exists i.\, 0 \leq i \leq n \wedge \\ &\qquad\quad (w_1 = v_i \vee \\ &\qquad\qquad ((v_i, w_1) \in \mathit{po} \wedge (w_1, v'_{i-1}) \in (\mathit{canSync}^{-1}\,;\, \mathit{po}^*)) \wedge \\ &\qquad\quad (w_2 = v'_i \vee \\ &\qquad\qquad ((w_2, v'_i) \in \mathit{po} \wedge (v_{i+1}, w_2) \in (\mathit{po}^*\,;\, \mathit{canSync}^{-1}))\} \end{aligned}$$

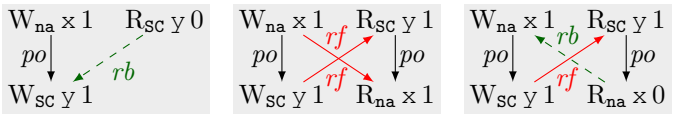
(noting that $r\,;\,s$ is the sequential composition of relations r and s, and r^{-1} is the inverse relation of r). That is, the path edge (v_i, v'_i) leads to the *po* edge (w_1, w_2) being put into *ppo* whenever:

- w_1 is equal to v_i, or it is *po*-after v_i and is the target of a *canSync* edge whose source is *po*-before or equal to the previous operation in the path (namely, v'_{i-1}), and
- w_2 is equal to v'_i, or it is *po*-before v'_i and is the source of a *canSync* edge whose target is *po*-after or equal to the next operation in the path (namely, v_{i+1}).

F. Ensuring correctness

The semantics of atomic operations in C, particularly the weakly consistent variants, is rather complex. Therefore, to ensure that our analysis is valid, we turn to automated tool support. We use the Alloy model checker [12], which has previously been successfully employed to validate other compiler mappings and optimisations in a weakly consistent setting [13], [3].

The C standard does not define the meaning of atomics on their own, but rather in terms of which executions of an entire program are allowed and which are not. An execution, in this context, is a set of runtime events with various dependencies between them. For example, the picture below shows the three 'candidate executions' of Program 2 from §I. W indicates a write event, R indicates a read event, and `na` means non-atomic.



The first candidate is the execution where the if-statement's test condition fails. Here, the *rb* ('reads before') edge indicates that the second thread's read of `y` is overwritten by the first thread's write to `y`. In the second candidate, the test condition succeeds and the new value of `x` is observed. Here, the *rf* ('reads from') edges indicate that the writes of `x` and `y` are observed by the other thread's read events. In the third candidate, the test condition succeeds but the old value of `x` is observed.

C allows the first and second candidate executions, but forbids the third. The mechanism for rejecting the third execution is the detection of a cycle made of *rf* edges between SC atomics, *po* edges, and *rb* edges. The precise rules that C uses to forbid executions are detailed by Lahav *et al.* [8].

Let us define a *buggy execution* to be an execution that is forbidden by C yet allowed by our implementation. The existence of such an execution would demonstrate that our implementation does not preserve enough of the program order. Characterising the executions that are forbidden by C is straightforward: they are the executions that violate at least one of Lahav *et al.*'s rules. Characterising the executions that our implementation allows is a little more subtle.

As discussed at the start of this section, the only source of memory reordering in our implementation is instruction reordering. Therefore, our starting point for characterising the executions that our implementation allows is simply SC. Shasha and Snir [14] characterise SC executions using the rule

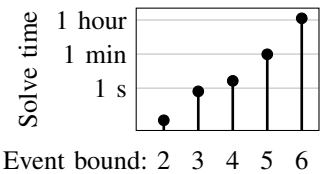
$$\mathbf{acyclic}(\mathit{po} \cup \mathit{rf} \cup \mathit{mo} \cup \mathit{rb}) \qquad \text{(Shasha–Snir)}$$

which states that there are no cycles made up of *po*, *rf*, *rb*, and *mo* edges. (The 'modification order', *mo*, is a relation between write events on the same location that represents the order in which the writes hit the main memory.) The rule works by rejecting executions in which data-flow (as captured by *rf*, *rb*, and *mo*) contradicts the program order. We weaken the Shasha–Snir rule by removing all the *po* edges that our analysis does not preserve, to obtain the following rule:

$$\mathbf{acyclic}(\mathit{ppo} \cup \mathit{rf} \cup \mathit{mo} \cup \mathit{rb}).$$

This rule has the same effect as Shasha–Snir applied to a program with a less constrained program order.

Alloy was able to confirm that there are no buggy executions with six events or fewer, for both methods of calculating *ppo*. The graph to the right shows that the time



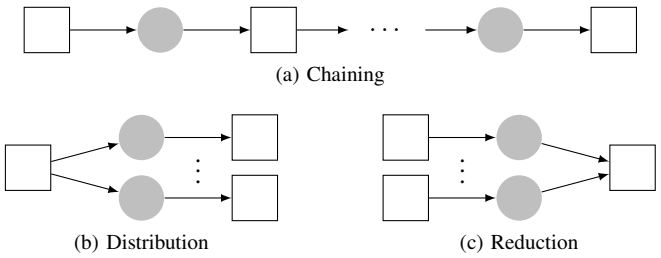


Fig. 5. The experiments we conduct for each data structure. Squares represent threads; circles represent data structure objects; arrows represent data flow.

taken for Alloy to deduce this result increases exponentially with the event bound. This is because Alloy casts the constraint-solving problem as a Boolean SAT query. Although a bound of six events appears small, note that Alloy's search space covers executions of *all* programs, so any bug that can be minimised to six events or fewer will be found. Experience indicates that most bugs related to weak memory can be minimised to between four and six events [15], so Alloy's result is a useful, if not completely watertight, validation of our method.

IV. Evaluation

We now evaluate how our global analysis compares to a thread-local analysis, on both SC and weak atomics.

In the theoretical best case, our global analysis can lead to an arbitrary speedup, if given a program such as

```
ast(&x1,1); ast(&x2,1);...; ast(&xN,1);
```

that consists of a series of atomic stores to different locations. A thread-local analysis would have to schedule all the stores sequentially, but our analysis can put them all in parallel, giving an $N\times$ speedup for arbitrary N. On the other hand, the worst-case speedup over a thread-local analysis is none at all, which would happen if every instruction in a program depends on its predecessor.

To give an indication of where the performance of our analysis on realistic programs lies between these two extremes, we evaluate three common data transfer patterns and three real-world lock-free data structures.

A. Experimental setup

We evaluate our analysis on the Treiber stack [5], a single-producer-single-consumer buffer [6] and the Michael–Scott queue [7]. We use versions of the stack and the queue that use weak atomics, due to Norris and Demsky [16].[2] The buffer only allows concurrent access between one writer and one reader, whereas the stack and the queue allow multiple readers and writers. These data structures are real-world examples of lock-free data structures and are part of the Boost library [17].

We evaluate these benchmarks on three common data transfer patterns: chaining, distribution and reduction. These patterns are designed to test one-to-one, one-to-many, and many-to-one relationships between threads, as shown in Fig. 5. For all experiments, we scale the thread count to test both the scalability of our analysis and the performance of the generated hardware.

[2]http://plrg.eecs.uci.edu/git/model-checker-benchmarks.git/

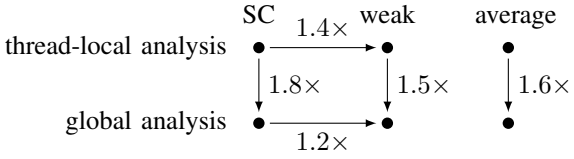


Fig. 6. Summary of speedups, averaged over all experiments

Our designs use LegUp 5.1's pure hardware flow, where each thread is instantiated as a hardware accelerator on FPGA. We use Quartus v15.0 to synthesise and place-and-route our designs to a Cyclone V SoC FPGA (5CSEMA5) with 32075 ALMs, 128300 registers, and 3970 Kb of RAM blocks.

B. Evaluating the performance of generated hardware

Figure 6 summarises the average speedups between the four design points in our evaluation, across all our experiments. The move from a local analysis on all-SC atomics to a local analysis on weak atomics was explored in our previous work [3] (but that implementation did not support CAS operations, hence the range of benchmarks was limited). We see that moving from SC atomics to weak atomics also benefits our global analysis, though the effect is slightly reduced. More significant speedups are obtained by moving from a local to a global analysis, even if programs only use SC atomics.

1) Global analysis versus thread-local analysis: Figure 7a shows the per-experiment speedup obtained, for the analysis that is sensitive to weak atomics, by switching from a thread-local to a global analysis.

We see that for the chaining experiments, the global analysis achieves an average speedup of $1.4\times$ for the buffer and $1.2\times$ for the queue, compared to a thread-local analysis. The buffer and the queue achieve good speedups because the global analysis is able to exploit memory reordering by analysing the interaction between a pair of push and pop routines across different threads. However, the stack sees no speedup because its small push and pop routines do not contain any opportunities for parallelism.

The distribution and reduction experiments achieve better speedups than the chaining experiments. This is because our global analysis is not only able to exploit memory reordering within routines but also to parallelise routines that access independent data structures. Hence, we see that the speedup scales proportionally with the thread count.

We remark that our global analysis actually leads to poorer hardware than a thread-local analysis on the buffer in the two-threaded reduction and distribution experiments (4% and 0.3% slowdown). Here, both analyses actually generate schedules with the same latency, but with different operations parallelised, and this leads to variations in the clock frequency.

2) Weak atomics versus SC atomics: The overall performance gained from performing the global analysis on weak atomics rather than just SC atomics is $1.2\times$. Figure 7b shows that this speedup is dependent on the benchmark. The buffer

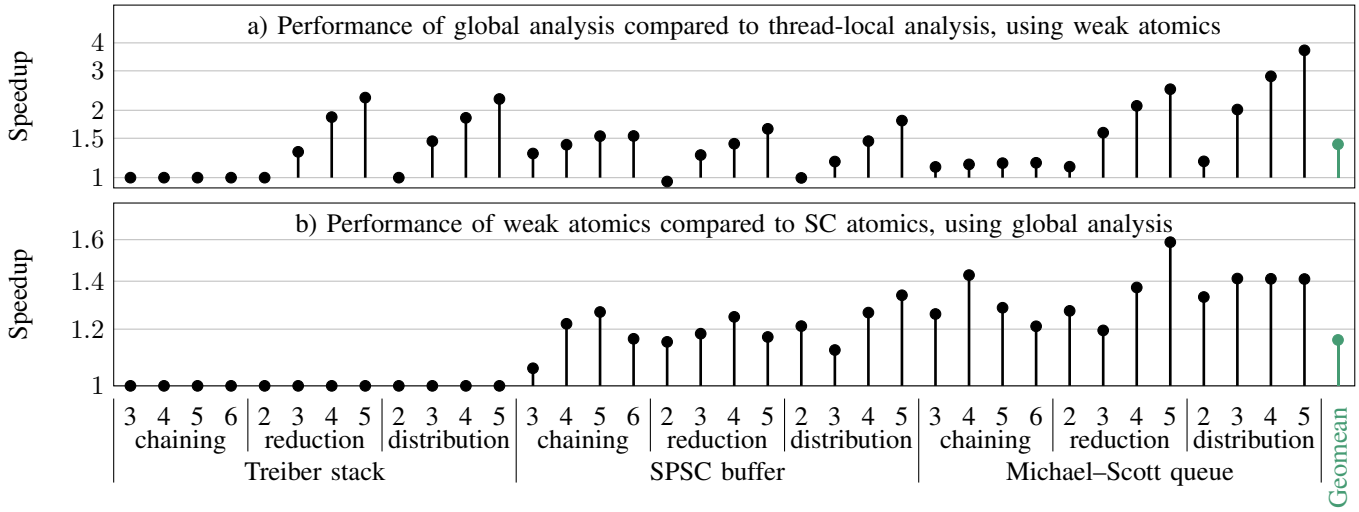


Fig. 7. Speedup (time) per benchmark. Data points are grouped by data structure, then by experiment, then by thread count.

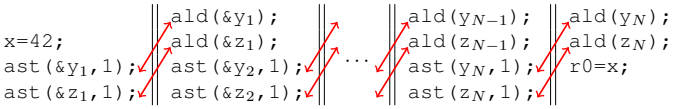


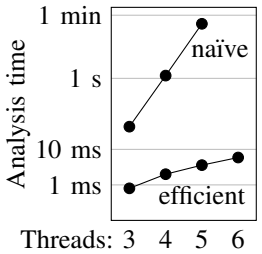
Fig. 8. A class of programs on which our analysis scales poorly, because the number of paths scales exponentially with the size of the program.

and the queue perform $1.2\times$ and $1.35\times$ better when the analysis is sensitive to weak atomics. These two benchmarks benefit from weak atomics since weak atomics allow more memory reordering within individual push/pop routines, compared to SC atomics. The stack, on the other hand, does not gain from the move to weak atomics, since weak atomics do not offer additional flexibility to its routines.

3) Resource utilisation: In most cases, our global analysis results in similar or slightly reduced logic utilisation compared to a thread-local analysis, on average saving under 1%. In the worst case, the logic utilisation increases by 6%, and in the best case, it decreases by 11%.

C. Evaluating the scalability of our analysis

The analysis time for all of our experiments ranges between 0.1 and 5.9 milliseconds, with a median of 0.4 milliseconds. This is using the efficient implementation of our analysis discussed in §III-E. To assess the impact of our efficient implementation, compared to a naïve enumeration of all paths, the graph to the right compares the time taken to analyse the Michael–Scott queue in the chaining experiment, with all atomics using SC mode. We see that as we scale the thread count, our efficient implementation outperforms the naïve implementation by several orders of magnitude ($9500\times$ at five threads). The naïve implementation cannot even complete the analysis at six threads, as it runs out of memory.



Nonetheless, the running time of our efficient implementation can still scale exponentially with program size, in the worst case. Figure 8 gives a pathological example of how this can occur. For each program obtained by instantiating the parameter N, there are 2^N primary paths from `x=42` to `r0=x` that must be explored. This is because there are two possible path choices for each stage in the chain, either via a `y`-variable or via a `z`-variable.

V. Related Work

Our previous work [3] exploited only a thread-local analysis when generating HLS scheduling constraints for atomics; the current work uses a global analysis instead. In most cases, our global analysis imposes fewer constraints than the local analysis, as seen in Example 1, since conservative assumptions have to made during local analysis. However, there also exist programs for which our global analysis imposes *more* constraints than the local analysis. This happens only in programs that access the same location using both an SC atomic and a non-SC atomic, and such programs are "not common" [8]. Indeed, we have used Alloy to verify that for all programs that do *not* mix SC and non-SC atomics on the same location, our global analysis never imposes more constraints than the local analysis. Alloy was able to prove this property for all programs with up to twenty operations in about a second.

Eliminating scheduling constraints in HLS is similar to eliminating fences in a conventional compiler. Our work is therefore related to that of Vafeiadis *et al.* [18] and Morisset *et al.* [19], who present (and prove correct) compiler optimisations that remove unnecessary fences from multiprocessor assembly code. Unlike our work, their optimisations exploit only a thread-local analysis.

Alglave *et al.* [20] present a whole-program analysis that determines where fences need to be inserted into a program to ensure SC behaviour when it is executed on weakly-consistent hardware. Their analysis is, like ours, based on enumerating inter-thread edges between conflicting accesses (similar to our

$canSync$ relation), and combining these with po edges to build paths. However, their analysis does not handle C atomics – all memory accesses are treated equally. As such, their analysis always guarantees SC behaviour. Our method, on the other hand, imposes only enough ordering to guarantee the behaviour specified by the programmer (which may be weaker than SC if weak atomics are used). Also, their analysis targets CPUs whereas our analysis targets hardware via HLS.

Crary and Sullivan [21] propose a new concurrent programming paradigm in which the programmer explicitly annotates their code with the program order edges that must be preserved; the compiler is then tasked with realising those requirements as efficiently as possible. The idea is to give the programmer more direct control over thread synchronisation than is possible through atomics and fences. Our work also involves preserving a subset of the program order edges, but where Crary and Sullivan's proposal is that this subset is entered by the programmer using a specialised language construct, we deduce it via a program analysis of an existing language.

GCC [22] and LLVM[3] both provide link-time optimisation (LTO), where the compiler has access to intermediate representation (IR) of all linked libraries and hence can treat the link-time program as a single compilation unit. If we adapted our implementation to insert fences rather than scheduling constraints, it may be possible to cast it as an LTO, and thus optimise atomics not just in HLS but in conventional compilers too. However, there are several challenges related to LTO such as the lack of one-to-one correspondence between the IR and source code, all the source code having to be compiled with the same compiler options, and the memory overheads of preserving the IR in an embedded environment.

VI. Conclusion

This work has proposed a global program analysis for multi-threaded C that determines which pairs of instructions within each thread must not be reordered by the HLS scheduler in order to implement the correct semantics for atomics. Our analysis can handle atomic loads, stores, and compare-and-swaps, and is sensitive to the consistency mode of each operation. We have provided a naïve description of our analysis, as well as an efficient way to implement it, and both versions have been checked for correctness against the C language standard via automated model checking. We have implemented our analysis by extending the LegUp HLS tool, and evaluated it on three real-world lock-free benchmarks. The results show that our global analysis yields an average whole-program speedup of 1.6×, with a minimal effect on resource utilisation. They also show that weak atomics are well-suited to our analysis, leading to an average speedup of 1.2× compared to SC atomics.

Overall, this work has demonstrated that global analysis of multi-threaded C programs can reduce the ordering constraints required to implement atomics correctly in HLS.

[3]https://llvm.org/docs/LinkTimeOptimization.html

Acknowledgements

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] Supplementary material is available on Zenodo, https://doi.org/10.5281/zenodo.1205395, and GitHub, https://constantinides.github.io/CATS-HLS/.

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

[3] 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.

[4] LegUp Computing Inc., "LegUp 5.1 Documentation." 2017, http://bit.ly/2D7VrQ0.

[5] R. K. Treiber, "Systems programming: Coping with parallelism," IBM Research, Tech. Rep. RJ5118, 1986.

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

[7] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *ACM Symp. on Principles of Distributed Computing (PODC)*, 1996.

[8] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, "Repairing sequential consistency in C/C++11," in *ACM Conf. on Programming Language Design and Implementation (PLDI)*, 2017.

[9] E. W. Dijkstra, "Cooperating sequential processes (1965)," in *The Origin of Concurrent Programming*, P. Brinch Hansen, Ed. Springer, 2002, pp. 65–138.

[10] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2011.

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

[12] D. Jackson, *Software Abstractions – Logic, Language, and Analysis*, 2nd ed. MIT Press, 2012.

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

[14] D. Shasha and M. Snir, "Efficient and correct execution of parallel programs that share memory," *ACM Trans. on Programming Languages and Systems (TOPLAS)*, vol. 10, no. 2, 1988.

[15] 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.

[16] B. Norris and B. Demsky, "CDSChecker: Checking concurrent data structures written with C/C++ atomics," in *ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[17] T. Blechmann, "Boost.LockFree," in *Boost C++ Libraries*, 2013, http://bit.ly/2qzHu8r.

[18] V. Vafeiadis and F. Zappa Nardelli, "Verifying fence elimination optimisations," in *Static Analysis Symp. (SAS)*, 2011.

[19] R. Morisset and F. Zappa Nardelli, "Partially redundant fence elimination for x86, ARM, and Power processors," in *Int. Conf. on Compiler Construction (CC)*, 2017.

[20] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl, "Don't sit on the fence: A static analysis approach to automatic fence insertion," *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 2017.

[21] K. Crary and M. J. Sullivan, "A calculus for relaxed memory," in *ACM Symp. on Principles of Programming Languages (POPL)*, 2015.

[22] T. Glek and J. Hubička, "Optimizing real-world applications with GCC link time optimization," *Computing Research Repository (CoRR)*, vol. abs/1010.2196, 2010.