Global Analysis of C Concurrency in High-level Synthesis

Nadesh Ramanathan *Member, IEEE,* George A. Constantinides, *Senior Member, IEEE,* and John Wickerson, *Senior Member, IEEE*

***Abstract*—When mapping C programs to hardware, high-level synthesis (HLS) tools reorder independent instructions, aiming to obtain a schedule that requires as few clock cycles as possible. However, when synthesising multi-threaded C programs, reordering opportunities are limited by the presence of *atomic operations* ('atomics'), the fundamental concurrency primitives in C. Existing HLS tools analyse and schedule each thread in isolation. In this article, we argue that thread-local analysis is conservative, especially since HLS compilers have access to the entire program. Hence, we propose a global analysis that exploits information about memory accesses by *all* threads when scheduling *each* thread. Implemented in the LegUp HLS tool, our analysis is sensitive to sequentially-consistent (SC) and weak atomics, and supports loop pipelining. Since the semantics of C atomics is complicated, we formally verify that our analysis correctly implements the C memory model using the Alloy model checker. Compared to thread-local analysis, our global analysis achieves a $2.3\times$ average speedup on a set of lock-free data structures and data-flow patterns. We also apply our analysis to a larger application: a lock-free, streamed, and load-balanced implementation of Google's PageRank, where we see a $1.3\times$ average speedup compared to thread-local analysis.**

I. Introduction

When mapping C programs to hardware, high-level synthesis (HLS) tools attempt to schedule all operations of a program into as few clock cycles as possible. They do so by identifying opportunities to reorder and parallelise independent operations without affecting program behaviour. However, when synthesising *multi-threaded* programs, the ability to reorder instructions is inhibited by the presence of *atomic operations* ('atomics).

Atomics are the fundamental concurrency primitives of the C language [2, §7.17], upon which more complex concurrency mechanisms such as mutual-exclusion locks, semaphores and various other lock-free data structures are built. An atomic operation must appear to be instantaneous to all threads and must also obey ordering constraints given by the C memory model [2]. Ramanathan *et al.* [3] showed that atomics can be implemented via HLS by injecting additional scheduling constraints within each thread. Following the standard practice of conventional compilers, memory constraints were determined locally, *i.e.* each thread was scheduled in isolation.

In this article, we argue that thread-local analysis of multi-threaded programs via HLS is overly conservative. HLS compilers have access to the entire program at compile-time, and thereby can consider the inter-thread memory synchronisations while scheduling individual threads. Hence, we propose a global analysis that exploits information about the memory accesses of *all* threads to determine, for *each* thread, all the pairs of memory accesses whose order must be preserved. We do so by enumerating all the possible pairs of memory operations through which the threads of an input program may synchronise with each other.

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 operations can be parallelised safely. The analysis proposed in this article correctly determines that both operations can be scheduled into the same clock cycle. Unfortunately, HLS compilers that only use thread-local analysis have to assume that there might be other threads concurrently accessing `x` and `y`, so these operations cannot be reordered safely.

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)

Here, the two stores in the left thread must not 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.

We implement our global analysis as an LLVM pass in the LegUp 5.1 HLS tool [4] and evaluate our implementation against thread-local analysis [3]. Since the C memory model is notoriously complicated, we formally verify our analysis to ensure the correctness of our generated hardware. We do so by using automated model checking via Alloy [5].

To evaluate the effects of our global analysis, we use a set of lock-free data structures and data-flow patterns as benchmarks. When we restrict our analysis to support *sequentially-consistent* (SC) atomics (the default consistency mode in

The authors are with the Department of Electrical and Electronic Engineering, Imperial College London, UK.
E-mail: {n.ramanathan14, g.constantinides, j.wickerson}@imperial.ac.uk

C), our results show that our global analysis leads to an $2.6\times$ average speedup, compared to thread-local analysis. Our analysis can also handle 'weak' atomics [2, §7.17.3], which impose fewer ordering requirements compared to SC atomics. Supporting weak atomics within our global analysis leads to a $1.9\times$ average speedup on our benchmark experiments, compared to use of weak atomics within thread-local analysis. Furthermore, our global analysis supports loop pipelining of atomics, which requires considering inter-iteration constraints. Enabling loop pipelining within global analysis provides an average speedup of $2.5\times$ on our benchmark experiments, compared to pipelined thread-local analysis.

To investigate the extent to which our analysis can be effective on larger applications, we explore Google's PageRank [6] as a case study. In addition to a baseline implementation of PageRank from an existing benchmark suite [7], we also apply standard lock-free streaming and dynamic load-balancing optimisations to this baseline. These optimised programs are handwritten and provided to both thread-local and global analyses for HLS generation. We show that, on average, global analysis improves the hardware runtimes of these implementations of PageRank by $1.3\times$, compared to thread-local analysis.

Article outline: Section II gives an overview of our key ideas by working through how our analysis applies to a simple program. Section III provides necessary background on high-level synthesis, and how concurrency works in C. Section IV describes the design and implementation of our global analysis, and how we ensure its correctness. Section V evaluates the effectiveness of our analysis on a set of small but representative benchmark programs. Section VI presents a case study on Google's PageRank.

Comparison to prior work: This article extends our conference paper [8] in two ways.

Firstly, we include a case study of Google's PageRank algorithm to demonstrate the effects of global analysis on a more complex and substantial application. We demonstrate that global analysis can identify complex patterns, such as streaming and load-balancing, effectively reducing schedule latencies of PageRank implementations. We also show that global analysis can generate all scheduling constraints for these programs within seconds, despite their scale.

Secondly, we extend our global analysis to support loop pipelining. Supporting pipelining requires path enumeration to consider iterations, which our prior work does not [8] since we assumed that memory operations do not overlap across iterations. In this article, we do not make this assumption. Hence, we discuss the necessary steps to support pipelining within global analysis and evaluate the effects of pipelined global analysis on our benchmarks, compared to pipelined thread-local analysis [3]. Furthermore, pipelined global analysis is critical in our case study since it improves streaming.

Companion material: Our companion material [1] includes benchmarks, Alloy models, and performance data.

II. Motivating example

In this section, we provide a more realistic program to intuitively showcase the benefits of global analysis.

Consider the three-threaded program in Fig. 1a. It has four global variables, of which two are atomic. Thread T0 consists of straight-line code performing four stores to independent locations, where the second (❷) and fourth (❹) accesses are atomic. Threads T1 and T2 both consist of a for-loop containing two loads, where the first load is atomic (❺ or ❼) and the second load is non-atomic (❻ or ❽) respectively. All atomic accesses in this example are sequentially-consistent.

This program represents a parallel programming idiom where a master thread distributes work to multiple threads via an independent sets of memory locations, or channels. Threads T1 and T2 attempt N times to read the message passed by T0 via a for-loop. If the respective flags of each channel are set, *i.e.* xr for T1 and yr for T2 (❺ and ❼), only then should the respective non-atomic data be read (❻ and ❽). Hence, the program assertion enforces that if the flag is set, then the non-atomic data written by T0 must indeed be available.

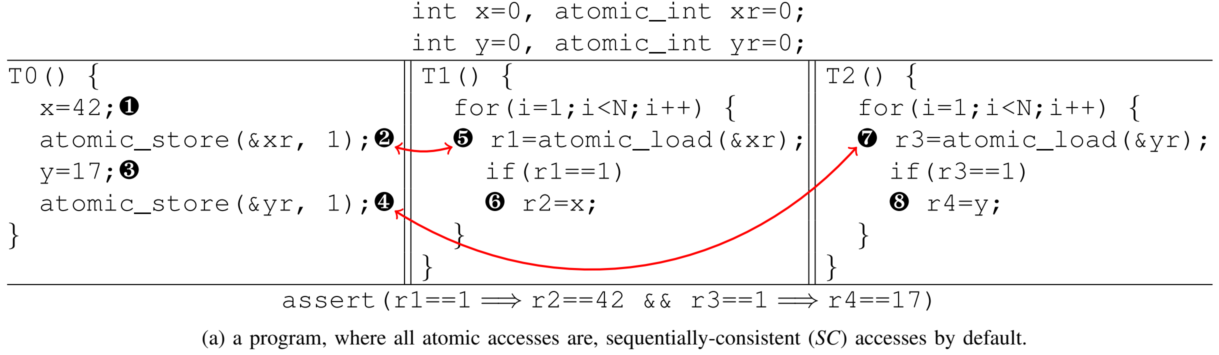
A. *Previous work: thread-local analysis*

Figure 1b shows an as-soon-as-possible (ASAP) schedule, without resource constraints, achieved by the constraints generated by thread-local analysis for each thread. Thread-local analysis is agnostic of inter-thread behaviour, thereby scheduling threads in isolation. Within T0, ❷ is atomic, so it must execute after ❶ and before ❸ and ❹. Similarly, ❹ is atomic, so it must execute after ❶, ❷ and ❸. Taken together, these constraints enforce serialisation of the four instructions in T0, and hence T0 takes four cycles to complete. In T1, ❺ is atomic, so it must execute before ❻, and in T2, ❼ is atomic, so it must execute before ❽. Hence, each iteration of T1 and T2 takes two cycles to complete.

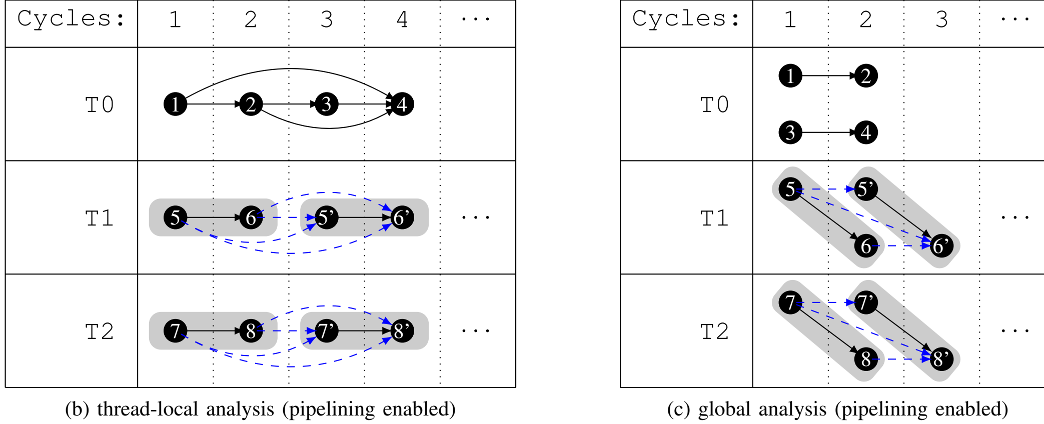
To pipeline the loops in T1 and T2, we must consider both inter- and intra-iteration dependencies. Loop pipelining aims to minimise the start times of consecutive iterations, referred to as initiation interval (II). Each operation within the loop repeats itself every II cycles, without violating any dependencies. When pipelining loops, thread-local analysis considers two consecutive iterations – this leads to the vertices ❺', ❻', ❼' and ❽' in Fig. 1b. Thread-local analysis does not allow atomic operations to be reordered with memory accesses from other iterations. Hence, ❺ and ❼ must execute before any operation in the following iteration, and ❺' and ❼' must execute after any operation in the previous iteration. We show these inter-iteration constraints using blue arrows. These constraints implicitly enforce iteration serialisation, inhibiting pipelining. Hence, the II and loop latency of these loops are the same and both loops still take $2 \times$ N cycles to complete.

B. *Our work: global analysis*

When we consider the memory accesses of the entire program, we can identify that there are two independent channels whose memory accesses actually need not be serialised. Let us consider all aliasing atomic accesses that can synchronise at runtime, *i.e.*: ❷ can synchronise with ❺ and ❹ can synchronise with ❼, as highlighted by the red bi-directional arrows in Fig. 1a. These atomic synchronisations cause other aliasing non-atomic operations to synchronise. If ❷ and ❺



(a) a program, where all atomic accesses are, sequentially-consistent (*SC*) accesses by default.



(b) thread-local analysis (pipelining enabled)

(c) global analysis (pipelining enabled)

Fig. 1. A three-threaded example to showcase the benefits of global analysis. Each thread is scheduled independently, and each operation is assigned to a clock cycle. The red arrows represent synchronisation opportunities of the input program, whereas the black solid and blue dashed arrows represent intra- and inter-iteration memory dependencies respectively. All memory operations within a shade belong to the same iteration.

synchronise, then every operation that precedes ❷ must be visible to every operation that follows ❺, and every operation that precedes ❺ must be visible to every operation that follows ❷. In our example, this means that ❶ must be visible to ❻. Similarly, if ❹ and ❼ synchronise, ❸ must be visible to ❽. Hence, the only pairs of memory accesses that must be executed in strict program order are: [(❶,❷),(❺,❻)] and [(❸,❹), (❼,❽)]. Consequently, as shown in Fig. 1c, global analysis schedules `T0` in two cycles (rather than four).

In the context of loop pipelining, symbolic unrolling of a loop iteration introduces additional synchronisation edges and paths between threads. In this example, ❷ can synchronise with ❺' and ❹ can synchronise with ❼'. These edges introduces additional paths: [(❶,❷),(❺',❻')], [(❸,❹), (❼',❽')], [(❶,❷),(❺,❻')] and [(❸,❹), (❼,❽')]. These paths include both intra- and inter-iteration memory constraints. Also, we must preserve order between aliasing operations across iterations: (❺,❺'), (❻,❻'), (❼,❼') and (❽,❽'). As shown in Fig. 1c, global analysis enables loops in `T1` and `T2` to execute with an II of one, taking $N+1$ cycles to execute instead of $2N$ cycles (by thread-local analysis).

III. Background

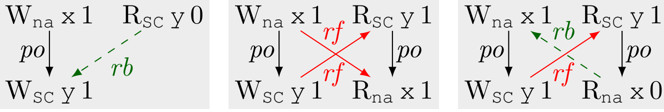
In this section, we present aspects of the C memory model (§III-A), HLS scheduling (§III-B) and related work (§III-C).

A. C memory model

The C memory model defines two types of memory operations: ordinary and atomic memory operations [2, §5.1.2.4, §7.17]. Ordinary memory operations can either be regular loads or stores. Atomic operations, *atomics*, are memory operations that: 1) must appear to be instantaneous to all threads and 2) are not allowed to be reordered with other memory operations within a thread. However, these properties ensure that all atomic accesses obey SC *i.e.* they are SC atomics. C11 also defines a set of weakly consistent C atomics, or *weak* atomics, that can be reordered with other memory operations under certain conditions.

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 disallowed. An execution is a set of runtime events with various dependencies between them. A runtime event is a memory event with four properties: event type, consistency mode, memory location and its value. There are two memory types: a write (W) or read (R). There are also four atomic consistency mode: sequentially consistent (`SC`), acquire (`ACQ`), release (`REL`) and non-atomic (`na`). Each event's written or read value is stated to the right of the location. Note that runtime events are not differentiated by thread, since executions are considered globally.

For example, the picture below shows the three 'candidate executions' of Program 2 from Example 1. This example has two locations: `x` and `y`.



In all executions, po ('program order') is the order of memory operations as stipulated by the programmer. 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 reads (values of reads from the same location must match with its latest writes). 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.* [9].

B. HLS scheduling

Scheduling is an important step in HLS where operations get assigned to clock cycles [10], without violating any dependencies. Typically, an input C program is represented as a control data-flow graph (CDFG) at the scheduling stage. 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 ($E_d \subseteq V_{\text{op}} \times V_{\text{op}} \times \mathbb{N}$). 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 these operations, if any.

One approach to implementing loop pipelining in HLS is modulo scheduling [11], [12]. Modulo scheduling computes a schedule where each operation is assigned to a particular cycle and repeated every II cycles, where II is the initiation interval. Typically, smaller IIs lead to faster execution but larger area.

The *system of difference constraints* (SDC) scheduling method [13] is a well-known method that is used by both industrial and academic HLS tools such as VivadoHLS [14] and LegUp [15]. Many scheduling requirements can be encoded as SDC constraints and powerful optimisations can be performed within this unified mathematical framework. Our work generates SDC constraints, where we focus on generating data dependencies (E_d). The SDC constraint that captures data dependencies is as follows:

$$\forall (v, v', dist) \in E_d : start(v') - end(v) \geq II \times dist. \quad (1)$$

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 least the loop initiation interval (II) multiplied by the loop dependence distance ($dist$). A dependence is *intra-iteration* when $dist = 0$, and is otherwise *inter-iteration*. Note that, in the absence of loop pipelining, the RHS of (1) reduces to zero.

C. Related Work

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.* [16] and Morisset *et al.* [17], 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.* [18] 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, 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, enforcing orderings via fences, whereas our analysis targets custom hardware, enforcing instruction orderings via scheduling constraints.

Crary and Sullivan [19] propose a new concurrent programming paradigm in which the programmer explicitly annotates their code with the program order edges that must be preserved. Our work also involves preserving a subset of the program order, but where Crary and Sullivan's proposal is that this subset is entered by the programmer via a specialised language construct, we deduce it via a program analysis of an existing language.

GCC [20] and LLVM[1] 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. Even though, LTO may have access to a single compilation unit, it may still be unclear as to which functions are to be executed, unlike in HLS. LTO also has its own challenges such as lack of correspondence between IR and source code, restricting all source code to be compiled with the same compiler options, and memory overheads of preserving IR.

Hsiao *et al.* [21] recently extended the LegUp scheduler to schedule all threads together. In comparison to our work: we employ a global analysis of all threads to devise one schedule for each thread, whereas Hsiao *et al.* devise one schedule that includes all operations in all threads. By considering multiple threads in a single CDFG, operations that share the same resource can be scheduled without resource contention. However, they must stall or pad loop bodies to ensure conflict-free memory accesses; our approach does not require stalls because our constraints guarantee that all possible executions of the input program will be correct.

IV. Method

In this section, we present a global analysis of multi-threaded programs with atomics that generates scheduling

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

constraints on a per-thread basis. That is, we use an *inter-thread* analysis to generate *intra-thread* constraints. Our analysis enumerates all the possible ways that the threads of a multi-thread program can synchronise. We use this information to decide which memory operations in each thread must not be reordered. We assume that all memory accesses are to on-chip memory elements and occur instantaneously, since current HLS tools [4], [22], [23] enforce direct mapping of memory without caches or write buffers. This assumption ensures that the only source of memory reordering is operation scheduling.

We implement our analysis in LegUp 5.1 [4]. We inject an LLVM pass between the Allocation stage and the Scheduling stage. All atomicity and consistency information of the memory accesses is available in the LLVM IR. We utilise LegUp's alias analysis [24], which based on Andersen points-to analysis, to determine which accesses are to the same memory location. Our analysis supports loads, stores, and compare-and-swap (CAS) operations, both on scalar variables and on arrays. C global variables and arrays are synthesised to registers and block RAMs respectively, where each shared memory construct is protected by an arbiter.

In this section, we discuss the following. In §IV-A, we describe the inputs to our analysis. In §IV-B, we explain how we explore all the possible ways that the threads of a multi-threaded program may synchronise. In §IV-C, we explain how we make our analysis sensitive to weak atomics. In §IV-D, we add support for loop pipelining. In §IV-E, we add support for atomic compare-and-swaps. In §IV-F, we present an optimisation that reduces the number of paths that must be enumerated. Finally, in §IV-G, we explain how we ensure the correctness of our analysis via automated model checking.

A. Input to our Analysis

The inputs to our analysis are sets of memory operations and relations over these sets. These sets and relations can be obtained from the LLVM IR and LegUp's alias analysis.

1) Sets: Let V_{mem} be the set of all memory operations for all threads in the entire program. Let V_{ld} and V_{st} be the sets of loads and stores, so $V_{\text{mem}} = V_{\text{ld}} \cup V_{\text{st}}$. (When we add CAS operations later, they will consist of two separate operations: one load and one store.)

Let V_{at} be the set of atomics, $V_{\text{at}} \subseteq V_{\text{mem}}$. Each atomic has one of four different consistency modes. Let V_{sc}, V_{acq}, V_{rel} and V_{rlx} be the set of sequentially consistent, acquire, release, and relaxed atomics, $V_{\text{at}} = V_{\text{sc}} \cup V_{\text{acq}} \cup V_{\text{rel}} \cup V_{\text{rlx}}$. Furthermore, we have $V_{\text{acq}} \subseteq V_{\text{ld}}$, $V_{\text{rel}} \subseteq V_{\text{st}}$ and $V_{\text{sc}} \subseteq V_{\text{mem}}$, because an acquire atomic must be a load and a release atomic must be a store. An SC atomic can be either a load or a store [2].

2) Relations: 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.

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

T0	T1	T2

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

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

Fig. 2. A three-threaded program that motivates the need to consider *paths* of *canSync* edges. `ast` and `ald` are short for `atomic_store` and `atomic_load` respectively.

B. Identifying instructions that must not be reordered

Our analysis begins by identifying pairs of operations that can cause threads to synchronise. It does this by defining the *canSync* relation, which connects any two atomic operations on the same location from different threads.

$$\textit{canSync} = (V_{\text{at}} \times V_{\text{at}}) \cap \textit{sloc} \setminus \textit{sthd}.$$

For instance, the *canSync* edges of our motivating example are given by the red 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. To ensure this, both *po* edges (❶,❷) and (❺,❻) must be preserved. Ensuring these intra-thread edges is sufficient to guarantee correct memory behaviour globally. Correct behaviour is guaranteed because these intra-thread ordering restrictions ensure safe interleavings between ❶,❷, ❺ and ❻ globally.

In general, we must consider not just isolated *canSync* edges, but *paths* of them, in order to handle programs like the one shown in Fig. 2. 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 a set, called *AllPaths*, of paths through the memory operations of the entire program. Here, 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 all of the following conditions hold:

$$\forall i.\, 0 \le i \le n \implies (v_i, v_i') \in \textit{po} \tag{2}$$
$$\forall i.\, 0 \le i < n \implies (v_i', v_{i+1}) \in \textit{canSync} \tag{3}$$
$$\forall i, j.\, 0 \le i < j \le n \implies (v_i, v_j) \notin \textit{sthd} \tag{4}$$
$$(v_0, v_n') \in \textit{sloc} \tag{5}$$
$$(v_0, v_n') \in V_{\text{ld}} \times V_{\text{ld}} \implies (v_0, v_n') \in V_{\text{at}} \times V_{\text{at}} \tag{6}$$

Condition (2) states that the path is a ordered list of $n+1$ edges from *po*. Condition (3) 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 4), since such paths can be

minimised or form cycles. This condition means that n cannot exceed the number of threads, and thus limits the length of paths that must be considered. Also, we are only interested in paths that start and end with accesses to the same location (Condition 5) 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 6). Non-atomic loads can be reordered because this reordering cannot be observed unless the program has a data race [9], and we can assume that, in C11 at least, a data race is a programmer error.

Finally, 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:

$$ppo = \{(v, v', 0) \mid \exists p \in AllPaths.\, (v, v') \in p\}. \tag{7}$$

We then provide all elements of ppo as SDC constraints to the scheduler.

As an example, in the absence of loop pipelining, there are only two paths in Fig 1a that satisfy Conditions (2) to (6): [(❶,❷), (❺,❻)] and [(❸,❹), (❼,❽)]. Hence, these four memory orderings need to be preserved, as shown in Fig. 1c: (❶,❷,0), (❸,❹,0), (❺,❻,0) and (❼,❽,0). Since these memory orderings are intra-iteration constraints, their dependence distance must be zero.

C. Adding support for weak atomics

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 §V, we show that the resultant reduction in scheduling constraints does indeed yield better-performing hardware.

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

$$canSync = ((V_{\text{rel}} \times V_{\text{acq}}) \cup (V_{\text{sc}} \times V_{\text{at}}) \cup (V_{\text{at}} \times V_{\text{sc}})) \cap sloc \setminus sthd.$$

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. Relaxed atomics do not introduce inter-thread synchronisation opportunities, and hence do not feature in this definition.

Figure 3a 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 [25]. It consists of two atomic locations, `x` and `y`, two release stores, and two acquire loads. If all the memory accesses were SC, the outcome `r1==r2==0` would be forbidden by C. To ensure that this outcome cannot happen,

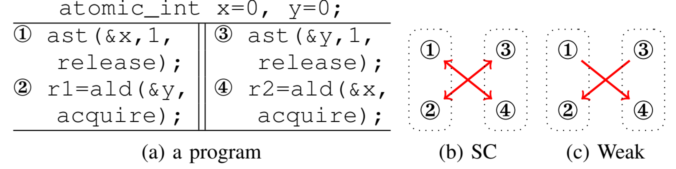


Fig. 3. The 'store buffering' programming pattern, and its $canSync$ edges under SC and weak consistency

our analysis would place $canSync$ edges as shown in Fig. 3b; this would lead to paths such as [(①, ②), (③, ④)] and hence both po edges are 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. 3c. No legal paths can be constructed with these edges, and hence the two instructions in each thread can be reordered.

D. Adding support for loop pipelining

Our input program may contain loops that can be pipelined. In such cases, memory operations could get reordered across iterations. Therefore, we must extend our analysis to not only consider paths with intra-iteration constraints, but also with inter-iteration constraints. In this article, we restrict all inter-iteration constraints to have a dependence distance of one. By doing so, our analysis can focus on preserving memory orderings between any two consecutive iterations, since an inter-iteration constraint that applies to two consecutive iterations inductively applies to subsequent iterations.

We can support loop pipelining within our global analysis with three additional steps. Firstly, we symbolically unroll each loop body by one extra iteration. Secondly, we extend the po relation to include this extra iteration, and also create one additional relation that is used by our analysis:

- *nite*, the 'next iteration' relation, which relates all memory operations in the current iteration to all memory operations of the next iteration.

We then define the orderings to be preserved by loop pipelining as *ppo-pipe*:

$$ppo\text{-}pipe = ppo\text{-}intra \cup ppo\text{-}inter \tag{8}$$

where

$$\begin{aligned} ppo\text{-}intra &= \{(v, v', 0) \mid \exists p \in AllPaths.\\ &\qquad (v, v') \in p \land (v, v') \notin nite\} \\ ppo\text{-}inter &= \{(v, v', 1) \mid \exists p \in AllPaths.\\ &\qquad (v, v') \in p \land (v, v') \in nite\}. \end{aligned}$$

Finally, we analyse the output of our global analysis to identify the inter-iteration memory constraints. All pairs of operations that are part of $AllPaths$ must be part of $ppo\text{-}pipe$. The $ppo\text{-}intra$ relation contains all the edges from the non-pipelined analysis, but assign them all with a dependence distance of zero. The $ppo\text{-}inter$ relation contains those edges that are also in $nite$; these have dependence distance of one.

We explain how our analysis supports loop pipelining by revisiting the example from Fig. II. First, we symbolically unroll the loop bodies of `T1` and `T2`. Then, we define

$$nite = \{❺, ❻\} \times \{❺', ❻'\} \cup \{❼, ❽\} \times \{❼', ❽'\}.$$

We then extend *po* to include (❺',❻'), (❼',❽') and *nite*. From this, our global analysis identifies the following new paths: [(❶,❷),(❺,❻')], [(❶,❷),(❺',❻')], [(❸,❹), (❼,❽')] and [(❸,❹), (❼',❽')]. Additionally, it also identifies inter-iteration aliasing constraints. This leads to:

ppo-intra = {(❶, ❷, 0), (❸, ❹, 0), (❺, ❻, 0), (❼, ❽, 0)}
ppo-inter = {(❺, ❻, 1), (❼, ❽, 1), (❺, ❺, 1), (❻, ❻, 1), (❼, ❼, 1), (❽, ❽, 1)}.

where the *ppo-intra* and *ppo-inter* are the black and blue arrows in Fig. 1c respectively.

E. Adding support for compare-and-swaps

We now explain how our analysis is extended to support atomic compare-and-swaps (CAS), which are central to many fine-grained concurrent algorithms [26]. 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.

For our analysis, we treat a CAS as a pair of accesses: an atomic load followed by an atomic store. If our analysis produces a *ppo* edge that constrains either the load or the store, we automatically map it to constrain the original CAS. This mapping guarantees the order of the CAS compared to all other operations in the thread. Additionally, CAS's load and store are guaranteed to execute in-order, since our analysis always preserves orderings of aliasing operations. The consistency modes of the load and the store are determined from the consistency mode of the original CAS. We map the consistency modes of a CAS operation into different load and store pairs: a *relaxed* CAS becomes a *relaxed* load and a *relaxed* store, an *acquire* CAS becomes an *acquire* load and a *relaxed* store, a *release* CAS becomes a *relaxed* load and a *release* store, an *acquire-release* CAS becomes an *acquire* load and a *release* store, and an *SC* CAS becomes an *SC* load and an *SC* store. CAS operations can be assigned different consistency modes for their success and failure cases; we only consider the success mode as it is always stronger [2, §7.17.7.4.2].

Guaranteeing the load and store order of a CAS is insufficient to guarantee these two operations happen instantaneously. A straightforward method of ensuring this in HLS is to perform the load and store while holding a lock [27]. However, using locks to implement atomics is inefficient because extra cycles are needed to acquire and release the lock. Locks also prevent a CAS operation from being reordered, even if it uses the *relaxed* consistency mode. Therefore, instead of using locks, we modify LegUp's RTL generator to support CAS operations directly in hardware. This modification ensures that no other memory operations to the same location can occur between the load and store of a CAS.

Figure 4 shows the generated memory architecture when two threads access a shared array. 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`) until the arbiter grants access to them. To perform a CAS on a RAM, a thread requires two consecutive cycles to complete an uninterrupted sequence of a read and write. 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. 4. Then, we modify each thread's state machine to pack the read and write into consecutive cycles. Finally, we implement comparison logic within the thread. Note that a CAS on a register does not require the `hold` signal since registers have zero-latency reads.

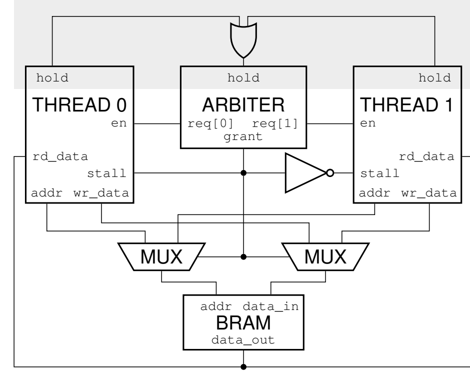


Fig. 4. A (simplified) circuit showing how CAS works for two threads. The shaded region indicates circuitry added by us.

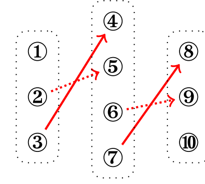


Fig. 5. Illustrating primary (solid) and secondary (dotted) *canSync* edges

F. 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 §IV-B. 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. 5 shows the shape of a program with three threads and four *canSync* edges. A naïve implementation of global analysis explores four paths with three edges each: [(①, ②), (⑤, ⑥), (⑨, ⑩)], [(①, ②), (⑤, ⑦), (⑧, ⑩)], [(①, ③), (④, ⑥), (⑨, ⑩)] and [(①, ③), (④, ⑦), (⑧, ⑩)]. The eight *ppo* edges from these paths are (①, ②), (①, ③), (④, ⑥), (④, ⑦), (⑤, ⑥), (⑤, ⑦), (⑧, ⑩) and (⑨, ⑩). Notice that the number of edges obtained from path enumeration is larger than the number of *ppo* edges obtained. Our optimisation aims to reduce these redundancies.

The two dotted *canSync* edges are secondary edges. For any path that passes through one or more of these secondary edges, there always exists a path between the same endpoints that

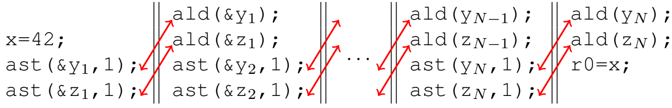


Fig. 6. A class of programs on which our analysis scales poorly, because the number of paths scales exponentially with program size.

does not pass through any secondary edges. More formally, we define the primary *canSync* edges as:

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

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 (3) to:

$$\forall i.\, 0 \leq i < n \implies (v'_i, v_{i+1}) \in \textit{canSyncPrimary}. \quad \text{(3a)}$$

Having calculated the set of primary paths, it remains to generate the *ppo* relation in a way that re-includes non-primary paths on a 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. 5, [(①, ③), (④, ⑦), (⑧, ⑩)] is the only primary path to explore. Thus, we have avoided exploring three other paths. Nonetheless, we can post-process this primary path with all secondary *canSync* edges to obtain the same *ppo*. For instance, from (②, ⑤), we can obtain (①, ②) and (⑤, ⑦). From (⑥, ⑨), we can obtain (④, ⑥) and (⑨, ⑩). Finally, from (②, ⑤) and (⑥, ⑨), we can obtain (⑤, ⑥).

More formally, we generate *ppo* as follows:

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

(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}).

Scalability: In practice, our optimisation can reduce the runtime of our global analysis on many real programs, as we discuss in §V-C and §VI-D. Nonetheless, the runtime of our optimisation could still scale exponentially with program size, in the worst case. Exponential growth can occur when our path enumeration must consider paths that encounter pairs of threads that have multiple primary *canSync* edges, which can result in a cascading effect on number of paths to explore. However, this behaviour may rarely occur since the program must be rather contrived or very large, both of which are unlikely in a HLS setting.

For example, Fig. 6 shows 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. There are two possible path choices for each stage in the chain, either via a `y`-variable or via a `z`-variable. Our optimisation does not eliminate any of these *canSync* edges, since they do not overlap. Such programs are good tests to understand scalability of global analysis, but are unlikely to be written in practice.

G. Ensuring correctness

The semantics of atomic operations in C, particularly the weakly consistent variants, is rather complicated. Therefore, to ensure that our analysis is valid, we turn to automated tool support. We use the Alloy model checker [5], which has been successfully employed to validate other compiler mappings and optimisations in a weakly consistent setting [28], [3].

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 previously, the only source of memory reordering in our implementation is by the corresponding instructions being reordered. Therefore, our starting point for characterising executions that our implementation allows is simply SC. Shasha and Snir [29] characterise SC executions using the rule

$$\textbf{acyclic}(\textit{po} \cup \textit{rf} \cup \textit{mo} \cup \textit{rb}) \quad \text{(Shasha and Snir)}$$

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 writes are executed. The rule works by rejecting executions in which data-flow (as captured by *rf*, *rb*, and *mo*) contradicts the program order (as captured by *po*). Intuitively, *rf*, *rb* and *mo* relate read-after-write, write-after-read and write-after-write events to the same location and Shasha-Snir asserts that these events must not form cycles with the order stipulated by programmer (*po*).

We weaken the Shasha and Snir rule by removing all the *po* edges that our analysis does not preserve, to obtain the following rule:

$$\textbf{acyclic}(\textit{ppo} \cup \textit{rf} \cup \textit{mo} \cup \textit{rb}).$$

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

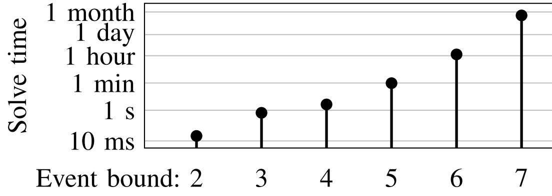


Fig. 7. Time taken for Alloy to verify our global analysis for up to a bounded number of memory events

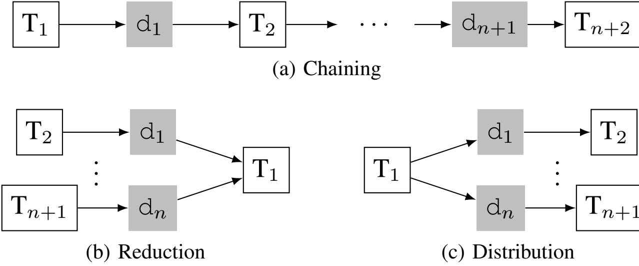


Fig. 8. The different data-flow patterns that we experiment for each data structure. Transparent squares represent threads and shaded squares represent data structure objects; arrows represent data flow.

Alloy was able to confirm that there are no buggy executions with seven events or fewer, both for the naïve method of calculating *ppo* (§IV-B) and for our more efficient version (§IV-F). Figure 7 shows that the time 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 seven events appears small, note that Alloy's search space covers executions of *all* programs, so any bug that can be minimised to seven events or fewer will be found. Experience indicates that most bugs related to weak memory can be minimised to between four and six events [30], so Alloy's result is a useful, if not completely watertight, validation of our method. Our verification of *ppo* also holds for loop pipelining of global analysis, since the C memory model does not distinguish events by iterations.

1) Comparison to thread-local analysis: In most cases, our global analysis imposes fewer constraints than the thread-local analysis, as seen in Example 1, since local analysis must make conservative assumptions. 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" [9]. Mixing SC and non-SC atomic accesses can inadvertently introduce bidirectional inter-thread synchronisations within global analysis. As such, these additional synchronisation opportunities increase the likelihood that global analysis generates more constraints compared to thread-local analysis. 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.

V. Benchmark Evaluation

In this section, we evaluate the performance of the hardware generated by global analysis, relative to thread-local analysis, on a set of experiments based on three data structures and three data-flow patterns. The data structures chosen are from high-performance software libraries, such as the Linux kernel [31] and Boost [32], and the data-flow patterns chosen are standard programming idioms for communicating across threads [26]. The aim of this section is to understand how our global analysis performs under scenarios that mimic real-world programs. We analyse how our analysis affects the performance (§V-A), resource usage of the generated hardware (§V-B) and how long our analysis takes to run (§V-C).

Data structures: We evaluate three data structures: the Treiber stack [33], a single-producer-single-consumer buffer [34] and the Michael–Scott queue [35]. We use versions of the stack and the queue that are implemented using weak atomics; these are due to Norris and Demsky [36].[2]

The stack and queue are implemented using linked lists, while the buffer is implemented as an array. The stack is last-in-first-out (LIFO) with only one atomic pointer, whereas the buffer and queue are first-in-first-out (FIFO) with two atomic pointers. The buffer only allows concurrent access between one writer and one reader, whereas the stack and the queue allow multiple readers and writers.

Data-flow patterns: We evaluate these data structures on three data-flow patterns: chaining, distribution and reduction. As shown in Fig. 8, these patterns test the one-to-one, one-to-many, and many-to-one relationships between threads:

- For the *chaining* pattern, shown in Fig. 8a, T_1 pushes to $\mathtt{d}_1$, and T_i pops data from $\mathtt{d}_{i-1}$ and pushes it to $\mathtt{d}_i$.
- For the *reduction* pattern, shown in Fig. 8b, T_2 to T_{n+1} push data to $\mathtt{d}_1$ to $\mathtt{d}_n$ respectively and T_1 pops data from all n data structure objects.
- For the *distribution* pattern, shown in Fig. 8c, T_1 pushes data to all n data structure objects, and T_2 to T_{n+1} pop data from $\mathtt{d}_1$ to $\mathtt{d}_n$ respectively.

Experimental setup: For all experiments, we scale the thread count, until $n = 8$, to test both the performance of the generated hardware and the time taken to run our global analysis. These experiments are provided as inputs to the LegUp HLS tool. 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.

A. Results: performance of generated hardware

Fig. 9 shows the speedups observed on the generated hardware across all our experiments. All subfigures compare the hardware generated by global analysis against hardware generated by theread-local analysis, under different settings.

Global analysis versus thread-local analysis using SC atomics: Fig. 9(a) shows the speedup gained by implementing our global analysis on SC atomics, compared to thread-local analysis. These speedups primarily differ by data-flow pattern. The speedups of the chaining pattern is $1.4\times$ whereas the speedups of reduction and distribution patterns are $2.3\times$ and $2.6\times$ respectively. The speedups of chaining do not scale

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

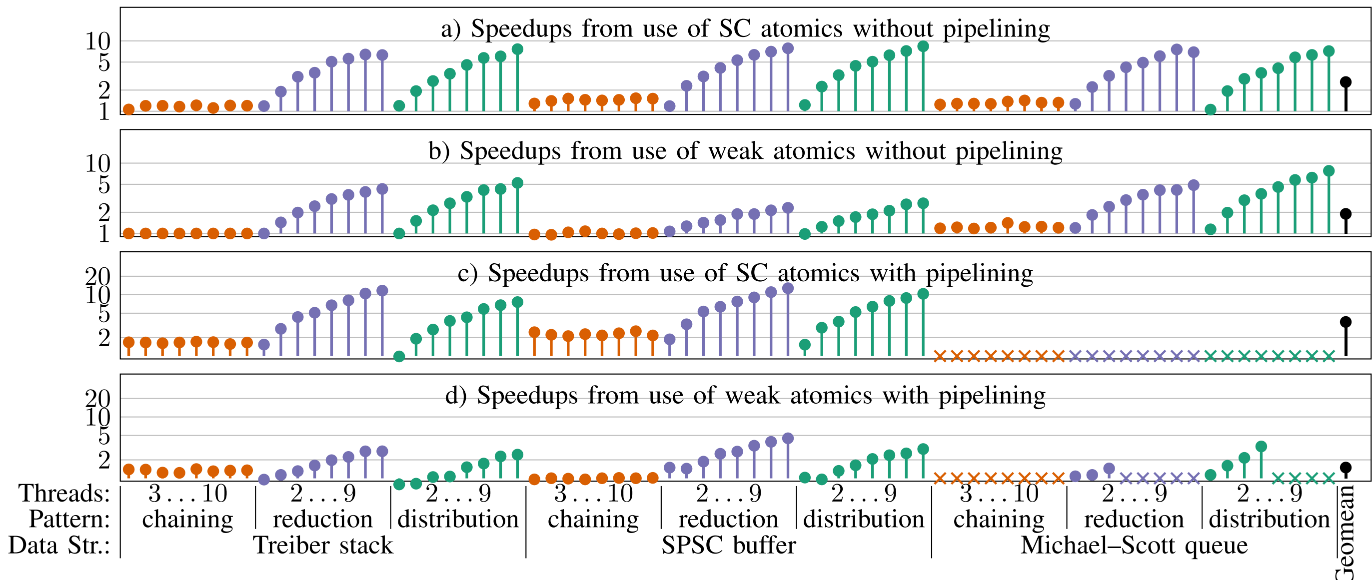


Fig. 9. Speedups (time) of hardware generated by global analysis compared to hardware generated by thread-local analysis per benchmark of our experiments, where subplots explore all possible configurations of type of atomics (SC or weak atomics) and pipelining capability. Datapoints are grouped by data structure, then by data-flow pattern, then by thread count. The rightmost bars show the average speedup across all data points for each barplot. Some data points are marked as $\times$ to reflect cases where *thread-local analysis* generated too many constraints that LegUp's modulo scheduling times out without a feasible solution.

with thread count, whereas the other patterns do since global analysis can identify independent routines.

Additionally, different data structures have have varying degrees of memory parallelism depending on their complexity of their routines. This can be identified via the chaining pattern, since each thread executes a fixed number of routines. The higher the complexity of a data structure, the more parallelism opportunities exist within its routine, and the higher speedups in the chaining experiments. Consequently, the queue offers the most parallelism opportunities, followed by the buffer and then the stack.

Utilising weak atomics via global analysis: Fig. 9(b) shows the speedups gained by global analysis, compared to thread-local analysis, when both these analyses are sensitive to weak atomics. Since weak atomics reduces the inter-thread synchronisation opportunities, global analysis can identify more instruction parallelism within threads. On average, global analysis is $1.9\times$ faster than thread-local analysis, when both use weak atomics. Similar to Fig. 9(a), the speedups are largely dependent on data-flow pattern. Hence, we see that global analysis is $2.3\times$ and $2.6\times$ faster than thread-local analysis for the reduction and distribution experiments, but only 7% faster for the chaining experiments.

Loop pipelining: Fig. 9(c) and (d) show the speedups achieved by loop pipelining within global analysis, compared to thread-local analysis, when using SC and weak atomics respectively. Thread-local analysis conservatively restricts the number of overlapping iterations to two iterations to guarantee correctness. On the other hand, global analysis enables memory overlappings beyond two iterations, depending on the data-flow pattern. Speedups can be more than $2\times$ for the reduction and distribution experiments, but not for the chaining experiments. On average, loop-pipelined global analysis is $3.6\times$ and $1.5\times$ for SC and weak atomics respectively, compared to pipelined thread-local analysis.

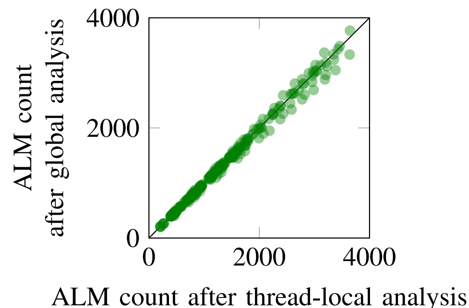


Fig. 10. Adaptive logic module (ALM) count generated by global analysis versus thread-local analysis for every data point in our experiments. Points below the diagonal show ALM savings when using global analysis.

Occasionally, thread-local analysis generates too many constraints, forcing LegUp's modulo scheduler to time out without a solution (marked as $\times$ in our plots). In comparison, global analysis never times out since it generates far fewer constraints than thread-local analysis. The number of memory constraints of thread-local analysis is either proportional to the use of SC atomics or the complexity of data structure. This is why the queue experiments are most likely to time out. The queue's routines consists of 22 memory operations, of which 20 operations are atomic including 5 CAS operations. In contrast, the buffer's and stack's routines only have 8 and 6 operations each.

B. Results: resource usage of generated hardware

Implementing global analysis imposes minimal resource overheads, compared to thread-local analysis, as shown in Fig. 10. The number of adaptive logic modules (ALMs) for designs generated by thread-local and global analysis is mostly close to the diagonal. Global analysis reduces schedule

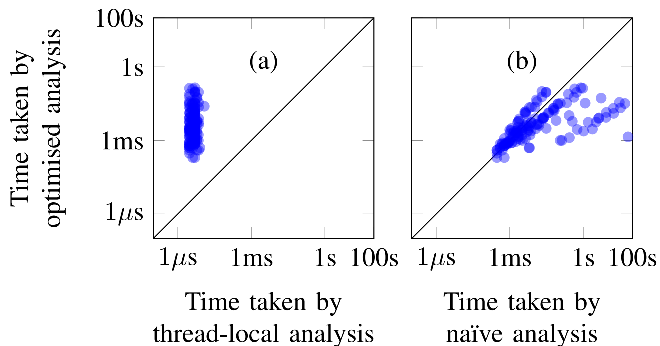


Fig. 11. Analysis times of three different analyses across all experiments. The y-axis of both plots are the time taken by our optimised global analysis and we compare it against time taken by thread-local analysis (left) and naïve global analysis (right) respectively. Points below the diagonal indicate cases where our optimised analysis is faster than the compared analysis.

latencies of our experiments, but packs more operations per scheduled cycle. These two factors affect resource usage inversely.

C. Results: Analysis time

In §IV-B and §IV-F, we discussed naïve and optimised path enumeration of global analysis respectively. Fig. 11 shows two scatter plots comparing the analysis times of our optimised global analysis against thread-local and naïve global analyses respectively. Fig. 11(a) shows that thread-local analysis always generates scheduling constraints faster than global analysis, by an average of $1000\times$. Fig. 11(b) shows that optimised global analysis is mostly faster than naïve global analysis, by an average of $3.3\times$.

Our optimisation works well when there are many secondary $canSync$ edges. It reduces the worst-case analysis time by up to $4700\times$. In particular, it is useful in the case of SC atomics, since using SC atomics tends to manifest more secondary $canSync$ edges, compared to weak atomics.

Utilising weak atomics naturally reduces the number of secondary $canSync$ edges. Most data points above the diagonal in Fig. 11(b) are programs with weak atomics. The maximum slowdown from using our optimisation is $5\times$, suggesting that programs with weak atomics may not require our optimisation. However, our optimisation can be seen as a precaution for programs that are fairly optimised, given that its time overheads are negligible compared to the time taken by hardware synthesis.

VI. Case Study: Google PageRank

In this section, we move beyond benchmarks, to show how more substantial lock-free programs can be synthesised efficiently with global analysis. We transform a multi-threaded C implementation of PageRank to incorporate standard hardware optimisations via lock-free data structures, including streaming and load-balancing. Subsequently, we demonstrate that global analysis can identify the memory patterns of these transformations, thereby permitting additional memory reorderings compared to thread-local analysis.

PageRank was first devised by Brin and Page [6] to improve web search engines by ranking webpages based on the graph of the web. A webpage is highly-ranked if the sum of the ranks of the webpages pointing to it is high. PageRank is interesting for several reasons. First, it is a well-known algorithm [37] that attracts a lot of acceleration work, especially on GPUs [7], [38]. Second, PageRank is in the form of a sparse matrix–vector multiplication (SpMV), which is a common pattern for graph algorithms. Third, PageRank's computation involves floating-point operations, providing a good balance between compute and memory intensity. Fourth, PageRank's workload cannot be evenly partitioned across threads, making it irregular.

In this case study, our baseline implementation is an OpenCL implementation of PageRank from the Pannotia suite [7]. In §VI-A, we minimally modified this OpenCL implementation into a pthreads implementation that is compatible with LegUp. However, this baseline is ill-suited to FPGA computation. So, in §VI-B, we introduce lock-free streaming to our baseline, by partitioning it into a streaming implementation that is capable of loop pipelining. Then, in §VI-C we introduce lock-free dynamic load-balancing to our streaming implementation. All lock-free data structures in this case study utilise weak atomics and all our analyses are configured to support weak atomics. Finally, in §VI-D, we show the hardware performance obtained by synthesising these versions of PageRank.

A. Introducing the off-the-shelf PageRank

```
1  int row[N]; int col[E];
2  float inR[N]; atomic_uint outR[N];
3
4  void *prk(int id){
5    for(int v = id*N/P; v<(id+1)*N/P; v++){
6      int start = row[v];
7      int end = row[v+1];
8      for(int edge=start; edge<end; edge++){
9        int u = col[edge];
10       add_float_atomic(&outR[u],
11         inR[v]/(end-start));
12 } } }
13
14 void main(){
15   pthread_t thds[P];
16   for(int i=0; i<P; i++)
17     pthread_create(thds[i], prk, &i);
18   for(int j=0; j<P; j++)
19     pthread_join(thds[j], NULL);
20 }
```

Listing 1. Pthreads implementation of PageRank, inspired by the Pannotia suite [7]. The `add_float_atomic` function in line 10 implements a floating-point atomic addition (details in the Pannotia codebase).

Listing 1 shows the C pthreads implementation of the PageRank algorithm, which we treat as our baseline. The only changes made are to spawn pthreads instead of OpenCL NDRange kernel (Lines 15–19) and to target all OpenCL local and global memories as C shared arrays (Lines 1–2).

The function `prk` is spawned as a thread with a unique thread identifier `id`. Each `prk` thread is a compute unit, which is a hardware circuit that can execute PageRank independently. We refer to the number of compute units as `P`.

PageRank consists of three input arrays and one output array. Array `inR` is the input rank and `outR` is the output

rank. The `row` and `col` arrays represent the input graph in the Compressed Sparse Row (CSR) format. `N` and `E` are the number of nodes and edges of the input graph respectively.

The algorithm consists of two nested loops. The outer loop iterates over all nodes (`v`) and the inner loop iterates over all neighbours (`u`) of a node. The input rank of each node must also be divided by its number of links before incrementing the output rank of `u` (line 11). Since several threads can update the same `u` simultaneously, atomic additions are required (line 10). Atomic floating-point additions are not supported in C natively. So, Pannotia implements this via a spin loop with an addition and a CAS.

Fig. 12a shows the hardware architecture that LegUp generates from the code in Fig. 1, when `P=2`. Each thread is a compute unit, and these compute units share the four arrays. This straightforward implementation of PageRank is synthesisable via HLS, but it is not optimised.

B. Partitioning PageRank into a streaming pipeline

To improve the performance of PageRank, we implement lock-free streaming on our baseline in §VI-A. We partition each compute unit into five pipelinable stages, each of which are a thread. As shown in Fig. 12b, these stages communicate with five lock-free buffers in a one-to-one streaming manner.

Partitioning PageRank into five stages required manual effort. Based on Listing 1, PageRank naturally partitions into three functional stages: fetching the CSR memory accesses (lines 5-9), dividing the rank by number of links (line 11) and atomically incrementing ranks (function call in line 10). The first stage (`Fetch`) has a small latency and the second stage (`Div`) can be easily pipelined. The third stage is complex since it implements the atomic addition. Consequently, we partition this stage into three smaller stages to improve pipelining: 1) the `FAdd` stage, which performs the floating-point addition; 2) the `CAS` stage, which performs the CAS; 3) the `Merge` stage, which merges new packets and packets with failed CASes.

C. Dynamic load-balancing via work-stealing

In previous implementations, an equal node ranges are allocated to each compute unit (line 5). This static allocation does not guarantee that compute units have equal workloads, since each node has a varying number of neighbours (line 8). One approach to reduce workload disparity is *work-stealing* [39].

Work-stealing requires use of a lock-free double-ended queue (or ‘deque’). We adapt the weakly consistent deque proposed by Lê *et al.* [40] for this study. A deque consists of two atomic pointers, protecting a non-atomic array. Each compute unit owns a deque that it can `pop` tasks from. When a compute unit’s deque is empty, then it attempt to *steal* tasks from other deques. PageRank has two properties that permit further optimisation. First, PageRank’s task is its node, so the non-atomic array and, consequently, all memory fences are not required. Second, PageRank does not spawn tasks, so we can eliminate the push routine and simplify other routines.

As mentioned earlier, we provide one deque per compute unit. Fig. 12b shows how work-stealing is implemented for `P=2`. Each compute unit is assigned a deque ($\mathtt{q}_1$ and $\mathtt{q}_2$).

TABLE I
DESIGN POINTS OF OUR PAGERANK IMPLEMENTATIONS AND THEIR BEST ACHIEVABLE THROUGHPUTS (MILLION EDGES PER SECOND).

Short name	Input	Analysis	Streamed	Stealing	Throughput
LocalBaseline	§VI-A	Local	no	no	6.7
GlobalBaseline	§VI-A	Global	no	no	6.7
LocalStreaming	§VI-B	Local	yes	no	14.6
GlobalStreaming	§VI-B	Global	yes	no	19.5
LocalStreamingWS	§VI-C	Local	yes	yes	20.3
GlobalStreamingWS	§VI-C	Global	yes	yes	30.0

Only the first stage, `Fetch`, needs to access the deques. $\mathtt{Fetch}_1$ and $\mathtt{Fetch}_2$ have exclusive pop access to $\mathtt{q}_1$ and $\mathtt{q}_2$ respectively. In the event that either ot them run out of work, they can steal work from the other deque. For larger Ps, the stealing pattern can be generalised to subsequent deques.

D. Evaluation

We evaluate PageRank on the same HLS tool and hardware as in our benchmark evaluation, described in §V We evaluate PageRank on the same graph used by Pannotia [7]: the DBLP co-author graph. We select the first 2^{10} nodes ($N = 1024$) and the corresponding edge list ($E = 5924$) to reside on RAM.

We evaluate six design points, all of which vary in terms of input code provided to LegUp HLS tool and the type of analysis we enable within LegUp to apply to the input code, as shown in Table I. We enable loop pipelining for all design points, by default. We scale the number of compute units (`P`) to understand their performance relative to area overheads, as shown in Fig. 13. The last column of Table I reports the best throughput achieved by each design point, regardless of number of compute units.

Baseline: *LocalBaseline* and *GlobalBaseline* show the results of an unmodified PageRank provided to LegUp, that is analysed by thread-local and global analyses respectively. The performance trends of both design points are so similar, that they overlap each other in Fig. 13. When PageRank is analysed off-the-shelf, global analysis does not offer much speedup.

Streaming: Streaming improves hardware performance significantly, since smaller pipelinable stages are better suited to FPGA computation. When `P` is maximised, *LocalStreaming* is $2.2\times$ faster than non-streamed implementations.

Global analysis improves the runtime of streamed PageRank implementations. Global analysis allows futher memory reorderings, compared to thread-local analysis, since it can identifies the one-to-one data pattern of streaming. This streaming pattern is similar to chaining in Fig. 8a. As we scale `P`, *GlobalStreaming* is always faster than *LocalStreaming*. When we maximised `P`, *GlobalStreaming* is $1.3\times$ faster than *LocalStreaming* and $2.9\times$ faster than *GlobalBaseline*.

Dynamic load-balancing: PageRank’s workload can not be evenly partitioned at compile time. To balance its workload dynamically, we implement work-stealing. At `P=2`, *LocalStreamingWS* is $1.2\times$ faster than *LocalStreaming*. The larger the `P`, the larger workload disparity across compute units. As we scale `P`, work-stealing provides better speedups. When we maximise `P`, *LocalStreamingWS* is $1.4\times$ faster than *LocalStreaming*.

When we apply global analysis to the streamed load-balanced PageRank implementation, *GlobalStreamingWS*, we

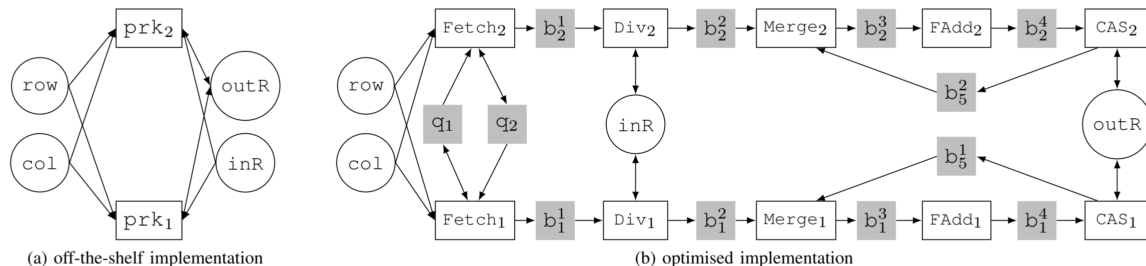


Fig. 12. Hardware diagram of the off-the-shelf and lock-free implementations of PageRank, synthesised by LegUp with either thread-local or global analysis. Transparent rectangles are threads that implement stages of a compute unit (subscripts are used to label compute unit). Circles are shared memory constructs. Shaded rectangles are lock-free data structures introduced to enable streaming and load-balancing, where b and q are lock-free buffers and deques respectively. Each subscript labels the compute unit ownership of each data structure and each superscript in the buffer represents its stage in the pipeline.

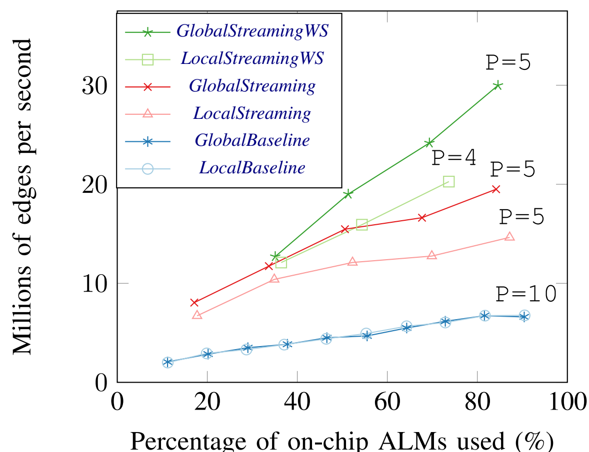


Fig. 13. Throughput versus hardware utilisation of all design points given in Tab. I. We scale the number of compute units (P) for each design point until it no longer fits on the chip.

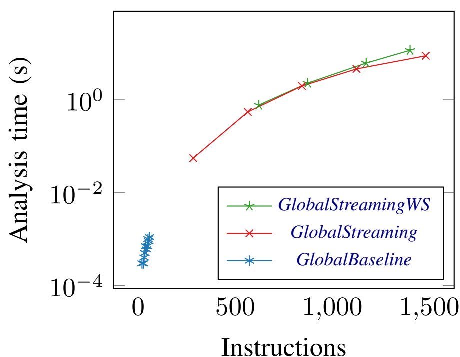


Fig. 14. Analysis times of our global analysis versus the number of instructions of our Pagerank implementations, as in Table I and Fig. 13.

further improve hardware performance for two reasons, compared to *LocalStreamingWS*. Firstly, global analysis can identify that compute units access multiple independent deques, similar to reduction in Fig. 8b. Secondly, work-stealing keeps all the streaming pipelines busy with work. Hence, when we maximise P, *GlobalStreamingWS* is 1.5× faster than *LocalStreamingWS*. Also, since global analysis requires marginally fewer resources, we can fit five compute units on-chip for *GlobalStreamingWS* but only four for *LocalStreamingWS*.

Analysis times: Fig. 14 shows the analysis times for global analysis given the number of memory instructions for the different PageRank implementations. We see that *GlobalBaseline* has very few memory instructions and it is the quickest to analyse. *GlobalStreaming* and *GlobalStreamingWS* have a much larger number of instructions than *GlobalBaseline*, since they implement complex lock-free data structures.

As we scale the number of compute units, P, we see that, in practice, our analysis times grow polynomially with instructions. At the P=5, our optimised analysis can generate all constraints within 11 seconds although these designs can have up to 30 threads, tens of thousands of po edges and hundreds of $canSync$ edges.

E. Summary

In this case study, we demonstrate that global analysis can improve memory scheduling by analysing complex patterns of lock-free data structures within the context of an important real-world application, namely Google PageRank. Results show that global analysis reduces the number of memory constraints generated, in comparison to thread-local analysis. On average, global analysis improves the performance across these PageRank implementations by 1.3×.

VII. Conclusion

In this article, we have proposed a global analysis for multi-threaded C programs that determines which pairs of instructions within each thread must not be reordered. Our global analysis can handle weak atomics as well as loop pipelining. We implemented our analysis in the LegUp HLS tool and evaluated it on a set of benchmark experiments, consisting of three data structures and three data-flow patterns. Overall, on average, we see that global analysis can achieve a 2.3× speedup, compared to thread-local analysis. Global analysis also enables us to manually apply standard HLS transformations to an HLS implementation of Google's PageRank, achieving a speedup of 1.3× compared to the original unmodified implementation. Having shown in this article that global analysis of multi-threaded programs can be valuable, we hope to encourage further investigation into the benefits of global analysis within other HLS contexts.

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, EP/K015168/1 and EP/R006865/1, an Imperial College Research Fellowship (Wickerson), and a Royal Academy of Engineering / Imagination Technologies Research Chair (Constantinides) is gratefully acknowledged.

References

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

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

[3] N. Ramanathan, J. Wickerson, and G. A. Constantinides, "Scheduling Weakly Consistent C Concurrency for Reconfigurable Hardware," *IEEE Transactions on Computers*, vol. 67, no. 7, pp. 992–1006, July 2018.

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

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

[6] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford InfoLab, Tech. Rep., 1999.

[7] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular GPGPU graph applications," in *IEEE Symposium on Workload Characterization (IISWC)*, Sep. 2013, pp. 185–195.

[8] N. Ramanathan, G. A. Constantinides, and J. Wickerson, "Concurrency-Aware Thread Scheduling for High-Level Synthesis," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2018, pp. 101–108.

[9] 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.

[10] 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.

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

[12] 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.

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

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

[15] 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.

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

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

[18] 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*, 2017.

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

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

[21] H. Hsiao and J. Anderson, "Thread Weaving: Static Resource Scheduling for Multithreaded High-Level Synthesis," in *Design Automation Conference (DAC)*, 2019.

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

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

[24] 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.

[25] E. W. Dijkstra, "Cooperating sequential processes (1965)," in *The Origin of Concurrent Programming*. Springer, 2002.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] "liblfds: A library of lock-free data structures," https://liblfds.org.

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

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

[34] K. Hedström, "Lock-free single-producer-single-consumer circular queue," 2014. [Online]. Available: https://bit.ly/2dbr8IK

[35] 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.

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

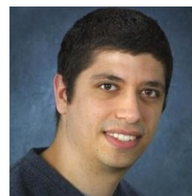
[37] S. Brin and L. Page, "The Anatomy of a Large-scale Hypertextual Web Search Engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, Apr. 1998.

[38] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.

[39] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999. [Online]. Available: http://doi.acm.org/10.1145/324133.324234

[40] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli, "Correct and Efficient Work-stealing for Weak Memory Models," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2013, pp. 69–80.

Nadesh Ramanathan received a PhD in Electrical and Electronic Engineering from Imperial College London in 2019. Currently, he is a Research Associate at Imperial College London. His research interests include reconfigurable computing, high-level synthesis, program analysis and formal verification. He is a Member of the IEEE.



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.

John Wickerson (M17, SM19) received a Ph.D. in Computer Science from the University of Cambridge in 2013. He is a Lecturer in the Department of Electrical and Electronic Engineering at Imperial College London. His research interests include high-level synthesis, the design and implementation of programming languages, and software verification. He is a Senior Member of the IEEE and a Member of the ACM.