Imperial College London
Department of Electrical and Electronic Engineering

High-Level Synthesis of Fine-Grained Weakly Consistent C Concurrency

Nadesh Ramanathan

October 2019

Supervised by Professor George A. Constantinides
and Dr. John Wickerson

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering of Imperial College London
and the Diploma of Imperial College London

Licence

Statement of Originality

I, hereby, declare that all of the work in this thesis is either my own or appropriately referenced.

Abstract

High-level synthesis (HLS) is the process of automatically compiling high-level programs into a netlist (collection of gates). Given an input program, HLS tools exploit its inherent parallelism and pipelining opportunities to generate efficient customised hardware. C-based programs are the most popular input for HLS tools, but these tools historically only synthesise *sequential* C programs. As the appeal for software concurrency rises, HLS tools are beginning to synthesise *concurrent* C programs, such as C/C++ pthreads and OpenCL. Although supporting software concurrency leads to better hardware parallelism, shared memory synchronisation is typically serialised to ensure correct memory behaviour, via *locks*. Locks are safety resources that ensure exclusive access of shared memory, eliminating data races and providing synchronisation guarantees for programmers.

As an alternative to *lock-based* synchronisation, the C memory model also defines the possibility of *lock-free* synchronisation via *fine-grained* atomic operations ('atomics'). However, most HLS tools either do not support atomics at all or implement atomics using locks. Instead, we treat the synthesis of atomics as a *scheduling* problem. We show that we can augment the *intra-thread* memory constraints during memory scheduling of concurrent programs to support atomics. On average, hardware generated by our method is 7.5× faster than the state-of-the-art, for our set of experiments.

Our method of synthesising atomics enables several unique possibilities. Chiefly, we are capable of supporting *weakly consistent* ('weak') atomics, which necessitate fewer ordering constraints compared to sequentially consistent (SC) atomics. However, implementing weak atomics is complex and error-prone and hence we formally verify our methods via *automated model checking* to ensure our generated hardware is correct. Furthermore, since the C memory model defines memory behaviour globally, we can *globally analyse* the entire program to generate its memory constraints. Additionally, we can also support *loop pipelining* by extending our methods to generate *inter-iteration* memory constraints. On average, weak atomics, global analysis and loop pipelining improve performance by 1.6×, 3.4× and 1.4× respectively, for our set of experiments. Finally, we present a *case study* of a real-world example via an HLS-based Google PageRank algorithm, whose performance improves by 4.4× via lock-free streaming and work-stealing.

Acknowledgements

I thank my PhD supervisor, Prof. George Constantinides, for his support in both guiding me towards producing novel, thorough and high-quality research as well as ensuring that I am focused and driven towards achieving my PhD goals. I also thank my second supervisor, Dr. John Wickerson, for imparting his technical insights on memory models as well as patiently encouraging me to communicate my research to the best of my abilities. I also thank Wiesia Hsissen, our group administrator, and Dr. Krysia Broda, the HiPEDS program manager, who have managed and advised me regarding college deadlines and funding requirements throughout my PhD.

I also thank my colleagues at the Circuits and Systems group, both past and present, People such as Felix Winterstein, Shane Fleming, Andrea Picciau, Ivan Beretta, Gordon Inggs, James Davis, Josh Levine, Ed Stott, Marlon Wijayasinghe, Junyi Liu, Xitong Gao, He Li and Matei Istoan have not only been good sounding boards but also have made the work environment at CAS vibrant and lively. I also would like to thank my HiPEDS cohort who have made PhD life at Imperial so much fun, due to their diverse backgrounds and technical experiences. So, thank you Ira Ktena, Emanuele Vespa, Christian Priebe, Panagiotis Garefalakis, Nicolas Miscourides, Nicolas Moser, Salvatore Diego Dipietro, Julian Sutherland, Konstantinos Boikos, James Targett, Antoine Toisoul and Yiannis Assael for keeping me sane. I also thank my Imperial hockey teammates for keeping my sanity. Finally, I thank all my other friends at Imperial, Cambridge University, Nanyang Technological University Singapore, and Uniten Malaysia as well as my extended friends and family who have been part of my complex education journey[1].

Most importantly, I also thank my brother, Ramesh Ramanathan, and my aunt, Valli Nadeson, for their unconditional care and support throughout my life. Last, but not least, I would like to thank my parents, N. Ramanathan and R. Leelambiga. None of this would have been possible without their sacrifices, struggles, patience, and vision.

[1]Extended thanks to Vimesshen, Ritesh, Pradeeban, Revathi, Arthur, Anushree, Lourene, Astrid, Prem, Tony, Arul, Maalini, Vani, Vairam, Vishneesh, Jana, Ghaja, Amandeep, Rubendran, Thevandran, Kanapathy, Sarveen, Ashwin, Kuhan, Gayathri and Priya.

Contents

List of Figures

List of Tables

Glossary

ALAP	as-late-as-possible
AOCL	Altera OpenCL
API	application programming interface
ASAP	as-soon-as-possible
ASSP	application-specific standard product
ASIC	application-specific integrated circuit
AXI	Advanced eXtensible Interface
BB	basic block
C11	2011 version of the C programming language standard
CAD	computer-aided design
CAS	compare-and-swap
CDFG	control data-flow graph
CSP	communicating sequential processes
CPU	central processing unit
DFG	data-flow graph
DSP	digital signal processor
FPGA	field-programmable gate array
GPU	graphics processing unit
HDL	hardware description language
HLL	high-level language
HLS	high-level synthesis
II	initiation interval
ILP	integer linear programming
IoT	Internet of Things
IR	intermediate representation

JMM	Java memory model
LLVM	low-level virtual machine
LUT	look-up table
MB	memory barrier
MCM	memory consistency model
MSQ	Michael-Scott queue
NP	nondeterministic polynomial time
OpenCL	Open Computing Language
OpenMP	Open multi-processing
POSIX	Portable Operating System Interface
PSO	partial store order
Pthread	POSIX thread
RAM	random access memory
RAR	read after read
RAW	read after write
RC	release consistency
RMO	relaxed memory order
RMW	read-modify-write
RTL	register transfer level
SC	sequential consistency
SDC	system of difference constraints
SDK	software development kit
SoC	system-on-chip
SPSC	single-producer-single-consumer
SpMV	sparse matrix vector multiplication
TLM	transactional level modelling
TS	Treiber's stack
TSO	total store order
WAR	write after read
WAW	write after write

WMM	weak memory model
WO	weak ordering

1. Introduction

High-level synthesis (HLS) is the process of automatically compiling a behavioural description of an algorithm into an RTL design [1]. The goal of HLS is to encourage design abstraction, which improves productivity of hardware experts and reduces the barrier-of-entry for hardware design for software programmers. Since C/C++ is one of the most commonly-used programming languages for both hardware and software designers, it is also the most popular HLS input. Despite its popularity, most HLS tools tend to focus only on synthesising features that are easily translatable and deemed useful. Hence, typically, advanced C features such as recursion, templates, dynamic memory allocation and explicit parallelism are not efficiently supported by HLS tools.

HLS support of C concurrency In this thesis, we explore the latest trend of HLS tools supporting C/C++ concurrency. Historically, HLS tools only supported sequential C programs. However, as the software world moves towards concurrency, concurrent programs are becoming mainstream due to their superior performance. As HLS tools are always seeking to expand their user-base, some of these tools are beginning to support concurrent C inputs. Intel and Xilinx support the synthesis of OpenCL via their SDKs [2, 3]. LegUp supports the synthesis of C pthreads [4], which is a C library extension that supports multithreading.

HLS treatment of concurrent C programs Although HLS tools are beginning to support concurrent programs, most of the tools still treat threads as a collection of independent sequential C functions. From a tooling perspective, this treatment is advantageous because concurrent programs can be supported incrementally by simply re-using techniques designed for sequential C programs. However, from a synthesis perspective, this treatment is a problem as it stifles the possibility of optimising concurrent programs as a whole. One instance of this problem is that memory scheduling of concurrent threads is done on a per-thread basis and that its memory constraints are based on the requirement of sequential C programs. This has two effects on the memory synthesis of concurrent threads. Firstly, only memory orderings between aliasing memory operations (*i.e.* memory operations to the same location) are preserved within a thread, since that is the requirement of a sequential C program. Secondly, since each thread is scheduled independently, memory scheduling of a thread is agnostic towards the memory behaviour of the rest of the program. These synthesis choices force HLS tools to either enforce lock-step memory synchronisation via barriers [5] or enforce mutual exclusion via locks [6].

Lock-based memory synchronisation Mutual exclusion enforces that only one thread can access a particular shared data structure at a given point in time [7]. Typically, locks, such as pthread mutexes [4], are used to enforce mutual exclusion [7]. A lock is a safety resource that only allows one thread to own it at a given point in time. If multiple threads access a shared data-structure simultaneously, then the program execution is racy and hence its behaviour is undefined. Although locks ensure safe shared memory access, lock-based programming has several drawbacks. Firstly, the use of locks may serialise shared memory accesses via different levels of granularity. At the finest granularity, individual memory accesses may be serialised. At a coarser granularity, memory accesses to a shared data structure may be serialised. At the coarsest granularity, all shared memory accesses can be serialised. Secondly, the use of locks may also lead to deadlocks. A deadlock occurs when, for various reasons, a thread does not release a lock that other threads are attempting to acquire, resulting in them waiting indefinitely.

Lock-free memory synchronisation As an alternative to mutual exclusion via locks, the C memory model defines the possibility for threads to synchronise without locks. Lock-free programming is made possible by fine-grained atomic operation ('atomics'). Atomic operations are indivisible memory operations that can synchronise with each other when they obey certain synchronisation orderings, as dictated by the C memory model. When these properties of atomics are obeyed, race-free and correct inter-thread memory synchronisation can be achieved without locks. The key advantage of lock-free programming is that we can avoid serialising memory accesses, which is enforced by mutual exclusion. However, current HLS tools either do not support atomics at all or support them inefficiently and discourage their use. To the best of our knowledge, only two HLS tools support the synthesis of C atomics. Intel's OpenCL compiler supports atomics but claim that they are expensive and they worsen performance [2]. LegUp HLS implements OpenMP atomics by wrapping locks around each atomic memory access, hence still resorting to the use of locks and defeating the fundamental reason of opting for atomics [8].

Goals and assumptions of this thesis In this thesis, we envision fine-grained C concurrency between concurrent threads within the boundaries of an FPGA chip. This vision is tailored to ensure that lock-free synchronisation across threads can be both non-trivial and inexpensive. Therefore, we assume that HLS tools compile C shared memory constructs directly onto on-chip memories without caches or write buffers. This assumption is in line with the current HLS compilation procedure of multi-threaded HLS tools, such as LegUp [8], Altera OpenCL [2] and SDAccel [3]. This assumption also provides the guarantee that memory operations are indivisible. Since indivisibility of atomics can be guaranteed via HLS, we explore the possibility of expressing their ordering guarantees within an HLS framework. Hence, the *primary goal* of this thesis is to devise an HLS-friendly method that can synthesise concurrent programs with atomics, which then enables the synthesis of lock-free concurrent programs. The mechanism to achieve this goal is to augment HLS memory scheduling to be sensitive to atomics. Naturally, there are several research questions (**RQ**) that manifest from our primary goal, all of which we address in this thesis:

RQ 1 How do we implement C atomics efficiently via HLS scheduling, instead of wrapping locks around each atomic access to ensure program correctness?

RQ 2 Since we have full control over HLS scheduling constraints, can we also support the weakly consistent C atomics, a range of operations that are less restrictive than the sequentially-consistent (SC) atomics, in an efficient manner?

RQ 3 Although we focus on scheduling threads individually, can these memory constraints be based on global analysis and would such an analysis provide speedups?

RQ 4 Since atomics can be implemented using HLS scheduling constraints, can we also support atomics in the context of loop pipelining?

RQ 5 What are the benefits of compiling atomics onto hardware via HLS and do our analyses provide any performance improvements?

We systematically address these research questions in our technical chapters (Chapters 3 to 6), where we list our original contributions at the beginning of each technical chapter. Then, in §7.1, we summarise our key contributions of our work and highlight the extent to which we solve our research questions.

1.1 Thesis outline

This thesis is organised in the following manner:

- In **Chapter 2**, we discuss the motivation of our work and the necessary background materials for this thesis, including thorough discussions on the critical aspects of HLS scheduling and the C memory model.
- In **Chapter 3**, we show how state-of-the-art HLS tools generate memory models that are too weak and must rely on mutual exclusion for any shared memory synchronisation. Then we formalise the HLS scheduling constraints for concurrent C programs and present how we can augment these constraints to allow for fine-grained synchronisation via atomics.
- In **Chapter 4**, we show why HLS memory scheduling of concurrent programs is conservative, since it is implemented on a per-thread basis. Then, we discuss why

and how global analysis can improve the memory scheduling of concurrent programs with atomics.

- In **Chapter 5**, we show how our analyses from Chapter 3 and Chapter 4 can be extended to support loop pipelining via additional HLS scheduling constraints and how these extensions can further improve performance.
- In **Chapter 6**, we present a case study on an HLS-based implementation of Google's PageRank algorithm. We improve the performance of this baseline implementation by introducing lock-free streaming and dynamic load balancing, where these performance improvements rely on all our analyses of previous chapters.
- Finally, in **Chapter 7**, we summarise our contributions and achievements of this thesis and highlight its potential and future possibilities.

1.2 Publications

The original contributions made in this thesis have been published as peer-reviewed conference papers and journal articles in the following publications:

1. **N. Ramanathan**, J. Wickerson, F. Winterstein and G. A. Constantinides, "A case for work-stealing on FPGAs with OpenCL atomics" in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)* (short paper).
2. **N. Ramanathan**, S.T. Fleming, J. Wickerson, and G. A. Constantinides, "Hardware Synthesis of Weakly Consistent C Concurrency" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA).*
3. **N. Ramanathan**, J. Wickerson, and G. A. Constantinides, "Scheduling Weakly Consistent C Concurrency for Reconfigurable Hardware" in *IEEE Transactions on Computers (TC), Volume 67, Issue 7, pp 992-1006, Jan 2018.*
4. **N. Ramanathan**, G. A. Constantinides, and J. Wickerson, "Concurrency-Aware Thread Scheduling for High-Level Synthesis" in *Proceedings of the 2018 IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM).*

1.3 Experimental Data

All experimental data presented in this thesis is hosted at [9] for reproducibility.

2. Background

In this chapter, we present our motivation and all necessary materials and concepts required to understand this thesis. We discuss the following:

- Firstly, in §2.1, we discuss the motivation to pursue the intersection between reconfigurable architectures and software concurrency.
- Then, in §2.2, we present the relevant HLS materials: such as tools and languages, abstractions, compilation processes and, most importantly, HLS scheduling.
- Additionally, in §2.3, we discuss the concept of memory models thoroughly, including its history, definitions, most importantly, the C memory model.
- Furthermore, in §2.4, we discuss the various concepts of parallel programming and the type of concurrent programs we are interested to synthesise via HLS.
- Finally, in §2.5, we discuss how we evaluate our methods, where we discuss a set of benchmarks and data-flow patterns that we group into experiments and also describe our experiment setups.

2.1 Motivation

To motivate our work, we present to current industry trends relating to reconfigurable computing and software concurrency. Hardware acceleration via custom computing is gaining traction due to its *energy efficiency* [11, 12]. Independently, as multi-core architectures become more pervasive, software written for computing is becoming *concurrent* to allow for parallel execution of algorithms [7, 13, 14]. In this thesis, we focus on exploiting these latest trends to extend the efficiency of hardware-accelerated computing. We discuss the following in the section:

- In §2.1.1, we discuss why the use of reconfigurable computing is prevailing.
- In §2.1.2, we discuss how software concurrency is becoming important.

Discussion on these two themes require reference to the CPU trends of the last 50 years. Hence, we refer to Rupp *et al.*'s collation in Fig. 2.1, which demonstrates the five important CPU trends: transistor, single-thread performance, frequency, power and core count.

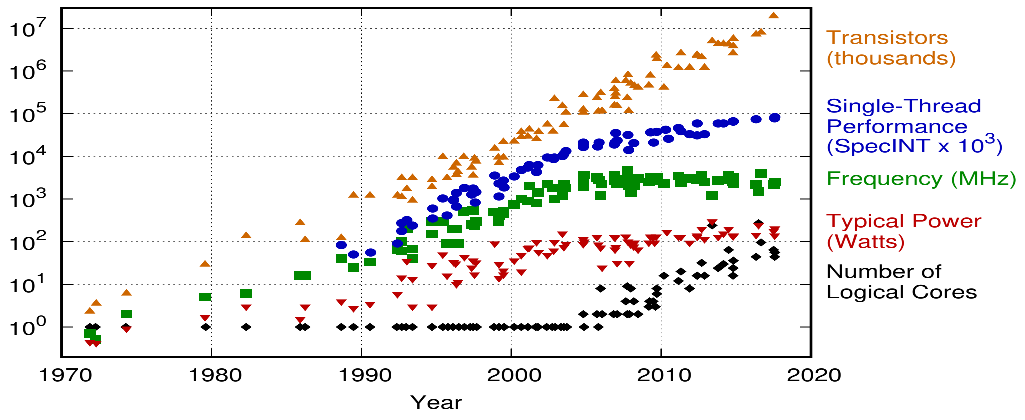


Figure 2.1: CPU trends for the last 42 years [10] (original data upto 2010 was collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten.)

2.1.1 Power wall leads to hardware customisation via FPGAs

Moore's law and Dennard scaling. Since the 1970's, the transistor count of a single chip has grown exponentially. This behaviour was predicted by G.Moore [15], where he stated that the transistor count, for a constant chip area, will double every two years. This prediction was mostly achieved by reducing the transistor's dimension, where with every new process generation the chip area halves for the same transistor count. Also, Dennard [16] observed that as we reduce the transistor dimension, a lower power budget is required for the same transistor count. In fact, for the same chip area, we can double the transistor count but keep the power budget constant. Hence, we see in Fig. 2.1 that frequency scaling and single-thread performance numbers increased at a faster rate than the power budget for the first 30 years.

Power wall due to leakage current. However, since the 2000s, the post-Dennard scaling era began. As we scale down the transistor's dimensions, the power supply voltage reduces to keep the electric field constant but the channel's leakage current increases [12]. Dennard did not account for this leakage current, as it was negligible until the supply voltage approached the transistor's threshold voltage. Hence, we see in Fig. 2.1 that the frequency and power budget plateaus after the 2000s and chip designers cannot continue to scaling the transistor count without increasing the power requirements of a chip. Hence, we are hitting a *power wall* in which only a subset of the chip area can be powered at a time, leading to an era of *dark silicon* [17].

Hardware acceleration reduces energy usage. As the power wall stifles microprocessor performance, chip designers are forced to think of more energy-efficient alternatives. The best way to reduce energy usage is via dedicated hardware acceleration [18], where performance-critical functions are implemented on hardware. There are two popular options for hardware acceleration [12, 18]: 1) full-custom logic such as ASICs or ASSPs, or 2) reconfigurable architectures. Full-custom logic, such as ASIC, ASSP or tightly coupled co-processors, offer the best energy efficiency and performance, since the function is optimised from the behavioural-level down to the layout-level [19, 20]. However, its design and verification efforts combined with its fabrication and non-recurring costs make it a high-risk proposition and unaffordable at low or medium volumes [21].

Reconfigurable computing is flexible and fast enough. Reconfigurable computing [11], on the other hand, implements arbitrary Boolean functions that can be dynamically reconfigured. Field-programmable gate arrays (FPGAs) are becoming more attractive as a hardware acceleration fabric. Over the last 30 years, FPGAs have also benefited from transistor technology scaling and have become increasingly large, more cost-effective and sufficiently energy-efficient for many applications [11, 22, 18]. Additionally, the inclusion of hardened logic, such as DSP blocks[1] and on-chip memories, within the FPGA fabric improves the performance of FPGA designs, closing the gap to GPUs [23] and ASICs [21].

Why are FPGAs sought after? The key advantage of the FPGA is reconfigurability, which has several benefits [11, 24]. Firstly, FPGA designs can be modified, upgraded and replaced at any point which is not possible with ASICs. Secondly, the time-to-market cost and development time is reduced significantly since the tape-out process of ASICs is eliminated. Thirdly, FPGAs better facilitate design space exploration, compared to ASICs, since design time to achieve a hardware implementation is reduced.

FPGAs are becoming more popular and readily available. As reconfigurable architectures increase in popularity, several significant investments in the technology have been made in recent times. In 2015, Intel acquired Altera, one of the largest FPGA vendors, for USD 16.7 billion [25] with the intent of expanding their market segments to low-power system-on-chip solutions, data centers and IoT. Amazon, via their Amazon EC2 F1 Instances [26], and Microsoft, via their Catapult infrastructure [27], are deploying FPGAs in a cloud environment to enable virtualised multi-hardware acceleration. One of the major disadvantages of FPGAs is that their cost per unit is high and constant with volume. However, deploying FPGAs in the cloud means that this cost can be amortised.

[1]A DSP is not to be confused with a DSP block. The former is a processor that is specialised for signal processing applications and the latter is a hardware block within the FPGA fabric that strictly performs arithmetic operations that are primirarily used for signal processing applications.

Design abstraction improves productivity. As the complexity of applications increases, higher levels of abstraction are required during the design process of reconfigurable computing. Abstraction is key as it eases the time and effort pressures during design synthesis and verification. Abstraction is the main reason for the introduction of *high-level synthesis* (HLS) [1, 28], where designs are described behaviourally in a high-level language (HLL) instead of being described at the Register Transfer Level (RTL). Despite abstracting away hardware idiosyncrasies, it has been shown that quality of generated HLS designs are comparable to RTL designs [29].

Summary Reconfigurable architectures allow for hardware acceleration beyond the tradition CPU power wall via hardware customisation. As reconfigurable architectures become more widely adopted, their application use cases and design complexity are bound to increase, requiring efficient design abstraction to express them efficiently. HLS plays a major role in design abstraction and hence, in this thesis, we explore synthesising an advanced feature of C – explicit fine-grained memory concurrency.

2.1.2 Exploiting parallelism is key to improving runtime performance

The scaling of a transistor's dimension contributes to a part of the single-thread performance improvements. Engineering efforts on how to utilise the additional transistor count also contributes significantly. According to Pollack's rule, microarchitecture advances lead to a further 1.4× performance improvement for every process generation [30]. Historically, these improvements are made by investing the available transistor count in exploiting two types of parallelism [12]: *instruction parallelism* and *memory parallelism.*

Instruction parallelism improves performance. Instruction parallelism can come in different levels of granularity [31]. Micro-architectural features such as out-of-order, superscalar and speculative execution enable multiple instructions to execute within the granularity of a single processor. Additionally, there are also architectures with specialised/vectorisation pipelines or co-processors that can execute multiple instructions or on multiple data simultaneously while the processor is busy executing other instructions. Finally, at the highest level of granularity, instructions can also be executed in parallel across multiple processors, which can appear in two forms: GPUs or many-core systems. However, such systems are generally tailored to suit applications that are '*single-instruction-multiple-data*' (SIMD) [31] in nature, where the same set of instructions is applied across a large chunk of data in lock-step manner.

Memory parallelism is equally important. Purely focusing on instruction parallelism is not always sufficient to improve performance. In order for the hardware units to be busy, the required data must arrive in a timely manner. Memory parallelism is typically achieved by allowing out-of-order memory executions and also non-blocking memory operations that mask access latencies [32]. As applications become more memory-intensive, memory parallelism is critical to improve overall performance at all levels of granularity.

Hardware acceleration requires the same attention towards parallelism. To achieve better performance via hardware acceleration, the target function/algorithm must be analysed for its inherent parallelism. In addition to exploiting parallelism, hardware acceleration can also improve the throughput of a function in the form of *pipelining* [33], where consecutive hardware stages/tasks can be overlapped with each other in time if they are not interdependent.

Single-thread performance saturates As seen in Fig. 2.1, the single-thread performance saturates slowly. Consequently, chip designers began investing transistors into duplicating these single-thread cores to form a *multi-core* architecture, as shown by the steady increase in core count since the 2000s. Thus, it is vital for software to be written in a concurrent manner so that algorithms can be truly parallelised on these architectures.

Software concurrency. Historically, the concept of concurrency existed even with the single-thread processor [13, 34]. A single-thread processor can only process on one task at a time but it frequently switches between tasks, *i.e. context-switching.* Context-switching provided programmers with the abstraction and illusion of parallelism, although the underlying hardware can only execute one task at a time.

Hardware concurrency. With the introduction of multicore architectures, *hardware concurrency* was made possible, where multiple tasks are be executed simultaneously on independent on-chip cores. This advancement allowed, or possibly forced [13], programmers to explicitly by partition their programs onto several tasks that can be distributed across multiple cores, or *multi-threading.* Multi-threading support for software ensures that concurrent programs can be executed in parallel by the underlying hardware.

Summary In this thesis, we are particularly interested in the synthesis of multi-threading via the POSIX threads (or *pthreads*) library [4, 35] via HLS. HLS tools can, now, compile concurrent software threads written in C into highly-customisable concurrent hardware units (which we refer to as *compute units*) on FPGAs. Although this advance enables instruction parallelism at the granularity of concurrent hardware threads, it still stifles memory parallelism to ensure program correctness by serialising memory synchronisation across threads via locks. Instead, we want to improve memory parallelism by compiling lock-free fine-grained C concurrency.

2.2 High-level Synthesis

High-level synthesis (HLS) is the process of automatically translating a behavioural description, written in a high-level language, into a register-transfer level (RTL) design, expressed in the form of a collection of gates (or netlist) [1, 28]. HLS has existed since the 1970s [28], though the first commercial HLS tools began to surface in the 1990s [1]. In this section, we discuss the following topics about high-level synthesis:

- First, in §2.2.1, we discuss the pros and cons of design abstraction.
- Then, we discuss how the various input HLLs and tools abstracts cycle accuracy (§2.2.2) and memory synchronisation (§2.2.3) descriptions at different levels of granularities.
- Next, we show how untimed C is compiled to RTL (§2.2.4) and then we discuss details of the compilation flow of LegUp HLS tool (§2.2.5).
- Finally, we discuss important details about HLS scheduling, including the SDC scheduling formulation, in (§2.2.6).

2.2.1 The pros and cons of design abstraction

High-level synthesis enables *design abstraction*, which allows the designer to describe the behaviour of an application rather than arduously implementing it directly in RTL. Design abstraction of HLS provides designers with several advantages [36, 37]:

- **Reduces design time.** Focusing on the behaviour of an application during the design process saves a lot of manual effort required during RTL design. The ease of use of HLS allows for rapid production of prototypes.
- **Improves simulation and verification.** Designs can also be functionally verified at the higher-level of abstraction instead of complex, error-prone and time-consuming RTL-level simulation and verification.
- **Enables design space exploration.** An application can be explored to fulfil various performance metrics (such as latency, throughput, area, energy). Additionally, instead of committing to several micro-architectural decisions early in the RTL design process, HLS tools via the injection of compiler directives allow for flexible micro-architectural explorations.
- **Reduces the barrier to hardware design.** Finally, since some HLS tools support pre-existing high-level languages (HLLs), such as C and Java, it opens up the possibility for software programmers to design custom hardware. It also increases the scope of applications targeted to hardware, since the range of pre-existing applications written in HLLs are large and re-usable.

Despite all of these advantages, the design abstraction of HLS also results in several

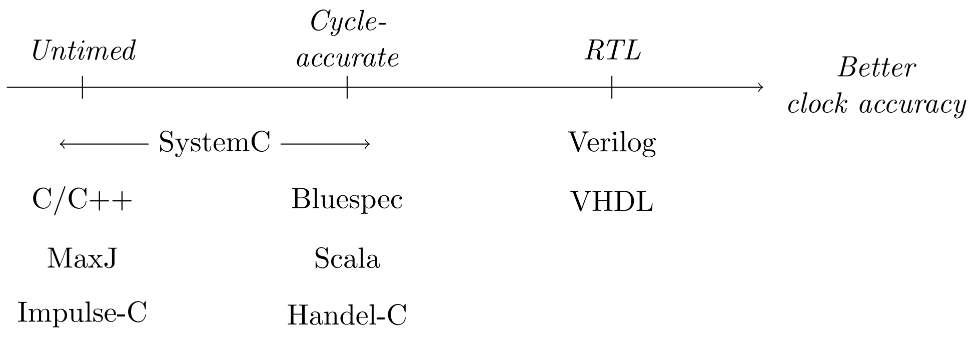


Figure 2.2: Classification of HLLs based on clock-level abstraction.

disadvantages to designers:

- **Optimising a design may require hardware expertise.** Although HLS enables designers to obtain an initial design quickly, fine-tuning a HLS-generated design requires hardware expertise and, possibly, in-depth knowledge of the specific transformations of a HLS tool. Non-experts may lack the know-how to inject the right directives or to refactor code for fine-tuning purposes.
- **Only a subset of a language may be supported.** Since the primary focus of HLS tools is design abstraction, they often limit the language features supported. Typically, only features that are deemed to be simple and useful are implemented by HLS tools. For example, typically, C-based HLS tools do not support dynamic memory allocation, recursion or templates and Java-based HLS tools do not support garbage collection. Hence, there are many situations in which the combination of a language and tool may not support an advanced feature.

2.2.2 High-level languages and their clock-level abstractions

Here, we discuss the various high-level languages (HLLs), their key features and the HLS tools that support them. We also discuss the clock-level abstraction provided by these HLLs, since we are interested in the HLS scheduling process. There are many ways to categorise HLLs. Fig. 2.2 shows the different types of HLLs organised in terms of their clock-level abstraction. There are two types of clock-level abstractions, besides RTL:

- *Untimed*: The most natural way of writing software algorithms is with no notion of timing. These HLLs typically require few modifications before the input is presented to a tool.
- *Cycle-accurate*: Unlike untimed HLLs, there are also HLLs that expose the system-clock and provide the notion of clock cycles to designers.
- *RTL*: RTL designs requires gate-level description, which demands complete timing, structural and functional specifications. While RTL designs potentially perform

best, it is arduous to re-time a circuit written in RTL.

The following HLLs are widely-used in both industry and academia:

- **C/C++** is the most widely-used HLL and is supported by several commercial and academic HLS tools, such as VivadoHLS [38], Catapult-C [39], ROCCC [40], LegUp [41], BAMBU [42], DWARV [43] and Kiwi [44]. C/C++ as a HLL is advantageous since this language has a large pre-existing userbase of programmers and codebase of applications. C/C++ inputs are untimed and require HLS tools to transform them into efficient RTL designs. We discuss this in §2.2.4.
- **Impulse-C** [45, Chapter 4] supports a C subset in the style of communicating sequential processes (CSP) [46], which is based concurrent processes that communicate via streams. Impulse-C is supported by CoDeveloper [47] and is untimed.
- **MaxJ** [48] is the only Java-based HLL, which is supported by the MaxCompiler. MaxJ enforces a streaming-like programming model that is then synthesised and mapped onto Maxeler's data-flow engine. This HLL is also untimed.
- **SystemC** [49] is a system design and modelling language tailored for hardware-software co-design. SystemC reduces development cost by combines electronic system-level design (ESL), which reduces serialisation of the design process, and transaction-level modelling (TLM) [50]. SystemC is supported by various commercial and academic tools as well, such as Catapult-C [39], Cynthesizer [51], CtoS [52] and VivadoHLS [38]. SystemC is unique because it supports both untimed and cycle-accurate inputs. An untimed SystemC input is simply a variant of C/C++. A cycle-accurate SystemC input can be described using `SC_CTHREAD`, which exposes the system clock by default. Furthermore, SystemC supports untimed, loosely-timed and approximately-timed transactions [53].
- **Handel-C** [54] also supports a C subset and additional software and hardware libraries that target reconfigurable fabric and also soft-processors on Xilinx and Altera FPGAs via the DK Design Suite [55]. This HLL is cycle-accurate, where its functions and expressions are evaluated in one clock cycle.
- **Bluespec** [56, 57] is a specialised HLL that is based on guarded atomic rules. Bluespec also has powerful static elaboration, interfaces, polymorphic overloading that makes it superior to Verilog/VHDL [58]. Bluespec is cycle-accurate since every guarded rule executes within a clock cycle [59], consequently providing designers with the notion of timing. Additionally, the system-clock is also exposable especially for clock-domain crossing.
- **Scala** [60] can be compiled to hardware via Chisel [61], a HLS tool developed by UC Berkeley. Since Chisel is embedded in Scala, it can support several high-level abstractions such as object-orientation, functional programming and parameterised

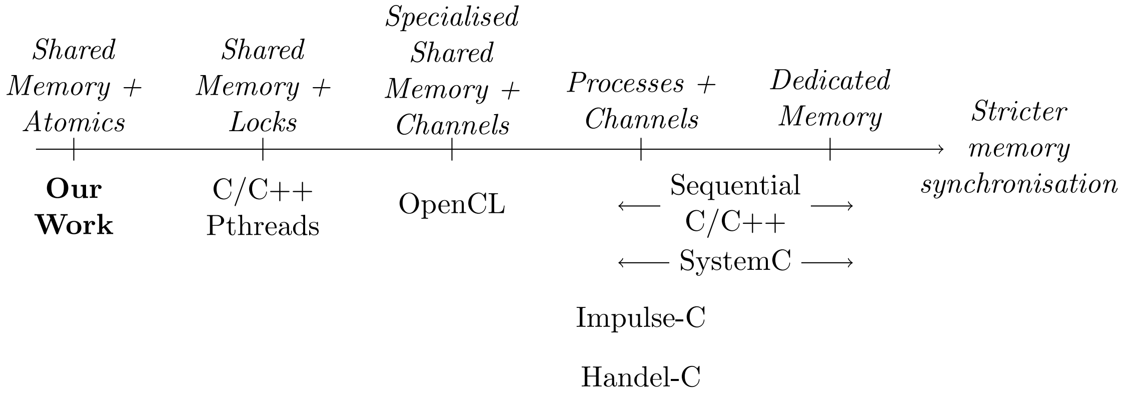


Figure 2.3: Memory synchronisation restrictions for C-based HLLs.

types. Chisel's hardware specification is similar to SystemVerilog and provides an abstraction that masks hardware modules as Chisel components [61].

2.2.3 Memory synchronisation for C/C++ via HLS

All C-based HLS tools support the synthesis of memory constructs. Fig. 2.3 classifies the memory synchronisation restrictions of C-based HLS.

Sequential C On the right-hand side of the spectrum, we have sequential C inputs that consist of C variables and arrays. Since there will only be one hardware unit, this unit is provided with dedicated memories, instead of shared memory, and hence there is no need for memory synchronisation.

Concurrent C processes with channels. To exploit further parallelism, HLS tools support the synthesis of multiple C functions that become independent parallel hardware units and these C functions are allowed to synchronise via dedicated communication channels. This is the de-facto standard for most HLS tools: independent C functions that communicate via dedicated vendor-specific channels. For example, VivadoHLS provides high-performance AXI streams across C processes [62], SystemC provides communication channels via TLM [53] and Handel-C and Impulse-C must infer communication channels.

Concurrent C processes with a memory hierarchy and channels. Although explicit parallelism is supported by HLS tools via synthesising independent C functions, this method is not unified from a programmer's perspective. Programmers specify independent C functions that adhere to specific communication APIs and then rely on HLS tools to generate a synchronised multi-hardware system. This programming model also stifles the possibility of expressing inter-thread memory synchronisation, a problem that was overcome by OpenCL. Intel's OpenCL SDK [2] and Xilinx's SDAccel [3] pioneered OpenCL HLS support for FPGAs. OpenCL provides a unified programming model that allows better expressiveness for memory synchronisation across threads via shared memory [5, §3.3]. Although OpenCL supports shared memory, its memory hierarchy is rigid and restrictive since it prohibits certain collections of threads from communicating with each other via memory scoping [5, §3.3.1]. This rigidity was put in place to optimise for specialised processors, where memory accesses are regular and execution is in lock-step.

Concurrent C processes with generic shared memory. Although OpenCL provides a unified programming model, its memory synchronisation is still limited due to the rigidity of its memory hierarchy and restrictions on memory synchronisation. Instead, HLS tools, such as the LegUp HLS tool [8], are now beginning to support the C pthreads library, which supports a generic shared memory space. Concurrent C programs that use the pthreads library allow arbitrary shared memory access, but it comes with a price as careful reasoning is required when synchronising across threads. The easiest way to synchronise via shared memory is the use of locks but they serialise shared memory access.

Our work Instead of using locks, we devise a method in which lock-free synchronisation can be achieved across threads via HLS. Hence, we can categorise our work as pushing the boundaries of HLS to the left of this memory synchronisation spectrum, as in Fig. 2.3.

2.2.4 High-level compilation of untimed C to RTL

In this section, we discuss the high-level compilation of untimed C descriptions into RTL designs. First, we present the process compiling C to hardware automatically (§2.2.4.1). Then, we present the general work flow of a HLS tool to achieve this (§2.2.4.2).

2.2.4.1 Compiling C to hardware automatically

Fig. 2.4 shows the compilation process of a C program into RTL [11, Chapter 7], which typically has four compilation stages. Also, dependent on tool support, the C code can comprise different features such as loops, branches, floating-point operations *etc.*

From C code to control-flow graph. First, the C code is compiled into a control-flow graph, as seen in Fig. 2.4(b). This graph consists of basic blocks as nodes and control-flow paths as edges. A basic block consists of straight-line code without no branches in, except at its entry, and no branches out, except at its exit.

Optimising CFGs Next, the compiler will perform optimisations and transformations on this CFG to improve the quality of the graph, such as clustering, combining basic blocks, expression optimisations and loop transformations. For example, Fig. 2.4(c) shows the clustering of basic blocks (via shades) to reduces the number of control-flow paths.

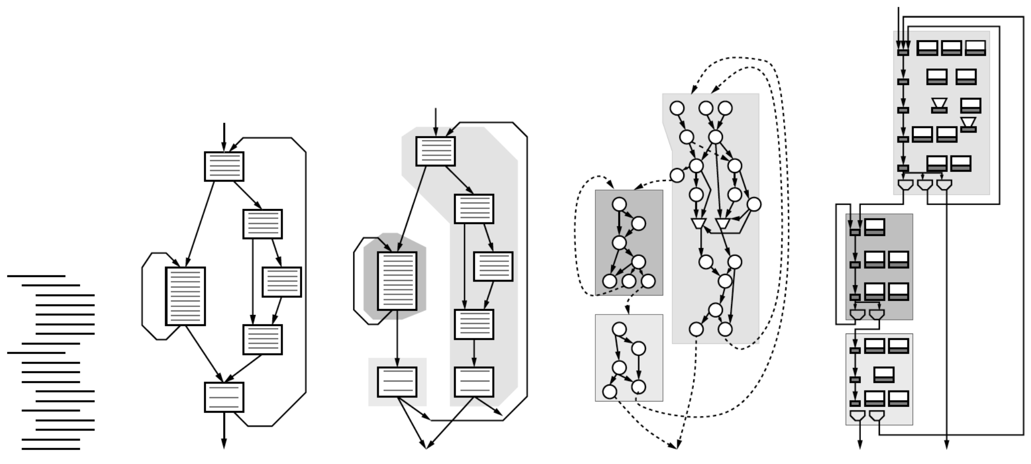

(a) C code (b) CFG of BBs (c) BB Clustering (d) BB to DFGs (e) Netlist

Figure 2.4: Overall flow for C compilation via HLS [11, Chapter 7]

Turning each basic block into data-flow graph. Once the CFG is finalised, each basic block is then converted into a data-flow graph (DFG), as seen in Fig. 2.4(d). This graph consists of operations (or instructions) as nodes and data dependence as edges. Data dependence can be distinguished into three types [11, Chapter 7]:

- *producer-consumer relationship*, where the output of one operation is required as input to another operation;
- *control dependence*, where instructions from one basic block are related to instructions in another basic block via control paths;
- *ordering constraints*, where the execution order of individual operations, such as memory operations, must happen in the order stipulated by the programmer.

From a control data-flow graph to RTL. Finally, the graph that contains both control-flow and data-flow, *i.e.* a control data-flow graph (CDFG), is transformed into RTL/netlist, as seen in Fig. 2.4(e). This final stage compromises four HLS steps: allocation, scheduling, binding and generation. Next, we discuss these stages.

2.2.4.2 Turning a CDFG into RTL

Once the CDFG is generated, a back-end compilation consisting of four steps is required: Allocation, Scheduling, Binding and RTL Generation. To achieve the best solutions, allocation, scheduling and binding should be executed together since they are interdependent. However, this approach becomes infeasible quickly for large problem sizes. Hence, the order in which these steps are implemented has significant impact on design quality [1].

Allocation During allocation, all operations in the CDFG must be assigned to a type of function unit/component. The exact number of components are typically defined as resource constraints. Components are selected from an RTL library, which must also provide the latency, area and other characteristics.

Scheduling During scheduling, all operations must be assigned to a clock cycle for execution. Scheduling is the step that introduces timing to an untimed C description.

Typically, the generated schedule must ensure that:

- all operations obey the CDFG dependencies; and,
- all design constraints, such as timing and resource constraints, are obeyed.

There are a range of methods that exist to find solutions for the various scheduling problems [63, Chapter 4]. We discuss these scheduling methods in §2.2.6.

Binding During binding, all operations must be connected to a hardware resource. The challenges of binding include the possibility of multiple resources that can perform an operation, multiple operations sharing the same resource at the different times and connectivity to customised and vendor-specific hardware resources such as buses, hardened multipliers and block RAMs [1].

RTL generation Finally, the RTL generation step is a straightforward process that translates the resultant design from the output of allocation, scheduling and binding into RTL. This RTL is then ready to be targeted to different logic synthesis tools, which is our case are the FPGA CAD tools.

2.2.4.3 C-based HLS tools

We considered several HLS tools as options to realise our work. We require a tool that supports the synthesis of concurrent C programs and also the ability to tap into the scheduling stage of this tool.

Vivado HLS Vivado HLS [38] is one of the most widely-used commercial HLS tool. This tool only supports the synthesis of sequential C programs, thereby eliminating the possibility of direct use of this tool. Indirect use of Vivado HLS is possible by synthesising each thread as a sequential C program and then interfacing these threads to shared memory via scratchpad memory frameworks such as CoRAM [64] or LEAP [65]. One example of this indirect use of interfacing Vivado HLS to LEAP was by Winsterstein *et al.* [66]. They analyse a sequential C program to identify sections of code that access independent heaplets, and interface them to a shared memory scratchpad that allows parallel accesses. We could not use their framework for two reasons. Firstly, their input program is a sequential C program. Secondly, despite being able to synthesise independent sections of code that may able shared memory at the same time, their work did not require tapping into the inner workings of Vivado HLS. They achieved the necessary hardware optimisations mainly via source-to-source code transformations and pragmas. For our work, we require intricate control of HLS scheduling.

OpenCL HLS There are two HLS tools that synthesise OpenCL programs: SDAccel [3] and Intel OpenCL [2]. SDAccel is a classic example of extending a HLS tool to incrementally synthesise concurrent C programs. Concurrent OpenCL threads/work-items within SDAccel are synthesised via Vivado HLS as independent sequential programs and interfaced to shared memory, without any support for locks. The Intel OpenCL tool, on the other hand, compiles a single hardware pipeline that executes several software threads [67, §1.2]. In fact, it recommends users to write OpenCL programs as a single software thread to improve compilation *i.e.* single work-item kernel [67, §1.3]. This tool also synthesises atomics, but discourage their use claiming that they are expensive and may lowers operation frequency [2, A.1.6]. Investigating the generated RTL design shows that atomics were implemented using a locking mechanism of a load-store unit [2, A.1.6]. Although these tools support the synthesis of concurrent C programs, their shared memory subsystems are not transparent, suboptimal and difficult to control. Additionally, understanding or extending the scheduling step within their tool flow is neither a straight-forward process nor encouraged.

Kiwi Kiwi [44] supports the synthesis of C# concurrency without scope for shared memory, since it is focused on accelerating custom arithmetics for scientific applications. Therefore, we can not explore the possibility of synthesising shared memory via Kiwi.

Why LegUp? We implement our analyses in LegUp [41] for several reasons. Firstly, LegUp support the synthesis of concurrent C programs via the pthreads library [8], instead of OpenCL via SDAccel or Intel OpenCL. Secondly, LegUp supports shared memory of C [68], which can be targeted on-chip and synchronised via mutexes. Hence, it allows us to leverage a synthesisable shared memory space that is generic and also allows us to compare our work against lock-based synchronisation. Thirdly, LegUp is open source and their scheduling is transparent, which allows us to implement our analyses as compiler passes that are tightly coupled to the original tool flow. Finally, LegUp's scheduling is constraint-based, which allows us to formalise its existing memory model and extend it to support the execution of atomics with the assistance of automated model checking. In the next section, we introduce LegUp in detail.

2.2.5 Understanding the LegUp HLS tool

LegUp HLS [41] is a C-based academic tool that is based on the LLVM framework, which is a modular and re-usable compiler and toolchain technology [69]. Since LegUp builds upon LLVM, it re-uses large parts of the LLVM infrastructure such as its toolchain, executables and passes. LegUp can target both pure hardware designs and hybrid software and hardware co-designs [8, 41, 68]. In this section, we discuss LegUp's general tool flow (§2.2.5.1), how we tap into their tool flow (§2.2.5.2), how LegUp synthesises pthreads (§2.2.5.3) and, finally, few more details necessary for this thesis (§2.2.5.4).

2.2.5.1 Executables

LegUp HLS has a three-stage compilation process, each of which is represented by an executable. These executables are as follows and perform the following tasks:

- `clang` is the front-end compilation that compiles C to LLVM Intermediate Representation (LLVM IR), which is the CDFG.
- `opt` is the middle-end compilation that performs optimisations and transformations on the IR to improve its quality. Additionally, LegUp implements its own hardware-oriented passes such as if-conversion, arbitrary bitwidth support, loop pipelining, array partitioning and pthreads support to prepare the IR for synthesis [6].
- `llc` is back-end compilation that converts the LLVM IR (CDFG) into RTL. Hence, this executable performs allocation, scheduling, binding and RTL generation and these stages are performed in sequence.

2.2.5.2 LLVM passes

Here, we discuss how we can influence the tool flow to insert our analysis. All the analyses, optimisations and transformations in `opt` and `llc` are implemented as LLVM passes on different levels of code hierarchy. There are four nested levels of class hierarchy in LLVM [69], which can be related to a CDFG's structure:

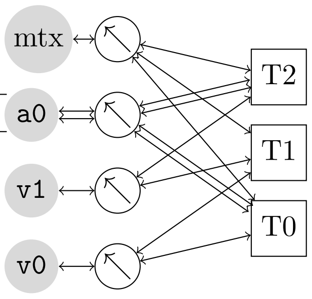
- The lowest level is an `LLVM Instruction`, which is a single LLVM IR instruction. An `LLVM Instruction` is an operation in a DFG.
- The second level is an `LLVM BasicBlock`, which is a collection of `LLVM Instructions` in a block of straight-line code. An `LLVM BasicBlock` is a DFG (or basic block) in the CDFG.
- The third level is an `LLVM Function`, which is a collection of `LLVM BasicBlocks` with control flow. An `LLVM Function` is a CDFG. C functions are compiled to `LLVM functions`.
- The fourth and highest level is an `LLVM Module`, which is a collection of `LLVM Functions`. An `LLVM Module` also contains `LLVM GlobalVariables`, which are the global variables of the input C program.

Hence, for us to tap into the LegUp tool flow, we simply need to augment the existing

```
volatile int v0,v1,a0[2];
pthread_mutex mtx;
```

T0	T1	T2
`mtx.lock()`	`mtx.lock()`	`mtx.lock()`
`int r0 = v0;`	`int r2 = v1;`	`int r4 = a0[1];`
`int r1 = v1;`	`int r3 = a0[0];`	`int r5 = v0;`
`mtx.unlock()`	`mtx.unlock()`	`mtx.unlock()`

(a) Sample access pattern



(b) LegUp 5.1

Figure 2.5: An example of memory architecture generated by LegUp Pthreads, where the shaded circles are memory elements and circles with arrows are arbiters.

LLVM passes or insert our own additional LLVM passes [70] at the appropriate stage of compilation and code hierarchy.

2.2.5.3 Synthesising pthreads

Throughout this thesis, we implement and evaluate our work via the LegUp 5.1 HLS tool [6], which we interchangeably refer to as *LegUp* from now on. LegUp supports the pthreads library and accepts multi-threaded C programs as inputs for both the pure hardware and hybrid flows [8]. LegUp also supports pthread mutexes [8], which provides the locking mechanism to achieve mutually-exclusive shared memory synchronisation. Recently, LegUp has also introduced streaming FIFOs based on these mutexes [68].

LegUp treats each pthread as a C function and compiles it as an LLVM function, which means each thread is an individual CDFG. LegUp then treats each CDFG independently and synthesises it to hardware. This is standard practice by HLS tools [2, 3, 71] that synthesise concurrent C programs, which has consequences on how memory accesses of concurrent programs are synthesised.

Generated memory architecture LegUp compiles all global variables and arrays as shared memory, under the pure-hardware pthreads flow. Each shared memory variable and array is instantiated as an individual on-chip register and block RAM respectively. Fig. 2.5(a) shows a sample program with two global variables (`v0` and `v1`), a global array (`a0`) and a pthread mutex (`mtx`). In this program, each thread accesses the mutex twice to form a critical section and within this section each thread performs two memory accesses.

Fig. 2.5(b) shows the memory architecture that LegUp generates for this program. We have four independent memory elements, where the array `a0` is a block RAM (symbolised by two-ported access arrows). Each memory element is protected by an arbiter, whose thread connectivity is based on the program's access pattern. Since each thread only accesses two memory locations, excluding the mutex, full cross-bar connectivity to the

Listing 2.1: Generated LLVM of `T0` in Fig. 2.5(a)

```
 1  define internal fastcc void @t0() #0 {
 2    br label %loop.i
 3
 4  loop.i:                                          ; preds = %loop.i, %0
 5    %1 = load volatile i32* getelementptr inbounds ({ { i32, i32, i32, i32, i32, {
        %struct.anon } } }* @mutex, i32 0, i32 0, i32 0), align 4, !legup_lock !1,
        !atomic_address !2, !data_width !1, !mutexName !3, !mutexType !4
 6    %2 = icmp eq i32 %1, 0
 7    br i1 %2, label %loop.i, label %legup_lock.exit
 8
 9  legup_lock.exit:                                 ; preds = %loop.i
10    %3 = load volatile i32* @v0, align 4, !tbaa !5
11    %4 = load volatile i32* @v1, align 4, !tbaa !5
12    store volatile i32 1, i32* getelementptr inbounds ({ { i32, i32, i32, i32, i32,
         { %struct.anon } } }* @mutex, i32 0, i32 0, i32 1), align 4, !legup_lock
        !1, !atomic_address !1, !data_width !1
13    %5 = tail call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([7 x i8]*
        @.str, i32 0, i32 0), i32 %3, i32 %4) #5
14    ret void
15  }
```

arbiters are not required. LegUp achieves this optimised connectivity via points-to analysis [68]. This analysis generates variable-local arbitration for each shared memory element *i.e.* LegUp generates local and shared-local memories. Memory elements that are only accesses by one thread are allocated local memories, memory elements that are shared between threads are allocated as shared-local memories and all memory elements that can be distinguished by alias analysis are allocated as one monolithic global memory.

LegUp 5.1's mutex locking and unlocking functions LegUp implements a mutex as a shared register and transforms its locking and unlocking functions to dedicated LLVM IR as a spinning load and a store respectively. These LLVM instructions are insufficient to implement locks correctly, since they can potentially race at LLVM level. LegUp annotates each mutex with a unique identifier, which the underlying hardware uses to avoid races [8, IV.B]. Listing 2.1 shows the generated LLVM IR from thread `t0` of the program in Fig. 2.5(a). The locking function is implemented as a spinning load, as seen in Lines 4-7. The current value of the mutex is loaded in an attempt to acquire the lock (line 5), which also has a unique identifier (`legup_lock`) attached to it to prevent races at hardware level. If the lock is acquired (check in line 6), then the critical section is executed (line 9). The unlocking function is implemented as a single store to the mutex location (line 12), which also has a unique identifier to prevent races at hardware level.

2.2.5.4 Additional details

Two additional details about LegUp:

- LegUp's alias analysis is critical to our work since it distinguishes whether two memory operations access the same location. This information manifests itself as ordering constraints during the scheduling stage in the `llc` executable. Previously, LegUp utilised LLVM's original alias analysis [72], but they have recently switched to Andersen points-to analysis [73], which allows for variable-local arbitration.
- LegUp only allows one basic block to be active during runtime within a CDFG [6]. This is a limitation of the tool itself, and our work is not affected since our ordering constraints are expressed as generic HLS constraints, as we will discuss in §2.2.6.1.

2.2.5.5 Limitations of LegUp

In the context of this work, LegUp limits us in a few ways:

- LegUp only supports shared-local memories via the pure hardware flow. In the pure hardware flow, shared-local memories allow threads to synchronise with each other via on-chip registers or RAMs. However, the pure hardware flow does not support any caches or off-chip memory accesses, such as DRAMs or processor memory. Therefore, our investigation of implementing atomics was restricted to on-chip memories.
- LegUp only allows one basic blocks to execute at a time. Our method does not rely on this execution model since we can generate constraints across basic blocks. However, LegUp ignores any inter-basic block constraints that we provide it, restricting reordering opportunities to only manifest within a basic block.

2.2.6 HLS Scheduling

Scheduling is an important step in the HLS flow that turns an untimed set of operations into a corresponding timed set of operations. Scheduling assigns a specific time (in clock cycles) for every operation in a CDFG, that respects the CDFG's data dependence and all resource constraints. All dependences and constraints must be obeyed to obtain a *feasible* schedule [63].

Classes of scheduling. Broadly, there are two classes of scheduling: time-constrained and resource-constrained scheduling [63, §5.1]. Both classes are generally solved by a mix of *as-soon-as-possible* (ASAP) and *as-late-as-possible* (ALAP) scheduling. Unconstrained scheduling, or ASAP scheduling, schedules each operation as soon as its predecessors in the DFG are scheduled. Latency-constrained scheduling, or ALAP scheduling, schedules each operation just before its successors in the DFG are scheduled.

Methods for resource-constrained scheduling. Resource-constrained scheduling is an NP-hard problem [63, §5.4]. There are two approaches to obtain a feasible schedule under resource constraints: exact or approximate methods. An exact method is ILP-based scheduling [74], which provides an optimum solution. However, the drawback of ILP-based scheduling is its time complexity can become intractable for larger problem sizes. Approximate methods such as force-directed scheduling [75] and list scheduling [76] can be more time-efficient, but may result in non-optimal solutions.

2.2.6.1 SDC scheduling

In this section, we focus on describing the *system of difference constraints* (SDC) scheduling method. SDC is a well-known method that is used by both industrial and academic HLS tools such as VivadoHLS [38] and LegUp [41]. A large set of scheduling constraints can be encoded as SDC constraints and powerful optimisations can be performed under this unified mathematical framework. These constraints can then be fed into scheduling optimisations such as ASAP, ALAP, longest path latency, expected overall latency and slack distribution. We are interested in SDC, because LegUp adopted this method [6].

2.2.6.2 SDC formulation

Now, we present the way in which SDC constraints are formulated. 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 ($dd \subseteq V_{\mathrm{op}} \times V_{\mathrm{op}} \times \mathbb{N}$). 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. In the absence of loop pipelining, this distance is simply zero.

Data dependences. In this work, we focus on the SDC constraint that captures data dependencies, which is formulated as [77]:

$$\forall (v, v', \mathit{dist}) \in dd : \mathit{end}(v) - \mathit{start}(v') \leq \mathit{II} \times \mathit{dist}. \tag{2.1}$$

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 the least the loop initiation interval (II) multiplied by the loop dependence distance (dist), where II is the number of cycles between the initiations of two consecutive loop iterations. A dependence is *intra-iteration* when $\mathit{dist} = 0$, and is otherwise *inter-iteration*. Note that, in the absence of loop pipelining, the RHS of (2.1) reduces to zero.

ASAP scheduling. Given a DFG with V_{op} and dd, we can achieve an ASAP schedule via the objective function:

$$\min \textstyle\sum_{v \in V_{\text{op}}} start(v)$$

where the fastest possible schedule of the DFG is obtained when the sum of the start times of all operations is minimised.

ALAP scheduling. We can also achieve an ALAP schedule by tweaking the above objective function to maximise the start times, instead of minimising them:

$$\max \textstyle\sum_{v \in V_{\text{op}}} start(v)$$

Resource-constrained scheduling Resource-constrained scheduling is generally an NP-hard problem. Hence SDC alleviates the NP-hardness of resource constraints by using a heuristic, which introduces a set of linear orders for each resource [78, §3.2.3]. Then SDC injects resource constraints based on this linear order. Let us say n is the number of operations that use the resource R and c is the number of instances of R. Given a linear order of operations that utilise R, $V^R = (v_1, v_2, \ldots, v_n)$, SDC resource constraints are expressed between every operation $v_i \in V^R$ and every operation that is c elements after v_i in V^R (if it exists), *i.e.* $v_{i+c} \in V^R$, as follows:

$$\forall i.1 \leq i \leq n - c : end(v_i) - start(v_{i+c}) \leq 0$$

where v_{i+c} can only execute after v_i. Cong *et al.* suggest a linear order, which is based on sorting operations according ALAP results, and using the ASAP results for tie-breaking[2], which encourages operation reordering and produces good-quality results [78, §3.2.3].

2.2.6.3 Modulo scheduling

Modulo scheduling [79], which is a well-known technique for loop pipelining, can also be implemented within an SDC framework [80]. LegUp implements loop pipelining via modulo scheduling [77]. Their implementation takes advantage of SDC to describe inter-iteration constraints. Since their objective is to achieve the minimum initiation interval, their implementation involves a backtracking mechanism to address a resource-constrained scheduling problem in a loop-pipelining context. From the perspective of generating the scheduling constraints for our work, we can simply encode inter-iteration data dependences as an SDC constraints, as presented in (2.1) of page 40. The dependence distance determines whether a constraint is intra- or inter-iteration (zero or a non-negative integer).

[2]LegUp implemented this tie-breaker based on program order rather than ASAP order. We fixed this issue to improve memory parallelism.

2.3 Memory models

The memory requirements of a single-processor system are simple: all memory accesses must be obey *coherence i.e.* all memory accesses to the same location must be executed in program order. Coherence does not restrict any reorderings between memory accesses to different locations *i.e.* non-aliasing memory locations. The simplicity of the coherence property encouraged micro-architectural advances and compiler optimisations to parallelise and reorder memory accesses safely and aggressively [32]. Unfortunately, these single-processor memory requirements are then replicated for multi-core architectures, which is insufficient to guarantee memory synchronisation without mutual exclusion.

A memory model, intuitively. In a multiprocessor environment where individual threads can access shared memory concurrently, a formal specification of the memory semantics is required *i.e.* a *memory model.* A memory model ensures that all processors access shared memory in a synchronised manner. Correct and efficient synchronisation is vital to ensure the programmability of a shared memory system, as writing concurrent programs may result in unexpected behaviours without these guarantees.

A memory model, more formally. A memory model defines the set of *legal executions* of a concurrent program. A concurrent program consists of the set of memory instructions or *operations.* These operations are compiled statically to a target hardware and executed dynamically as memory events. Depending on the input program, there can be exponentially many permutations in which these memory events can be executed at runtime. From this set of all possible executions, the memory model identifies the set of *legal executions* based on a set of rules over memory events (or event relations). An execution is deemed to be illegal if it forms a cycle over the rules defined by the memory model.

Where are memory models enforced? Historically, memory models were first enforced on hardware architectures. These models provide the programmers with memory execution guarantees of its assembly code. As hardware memory models became a necessity, programming languages also began to describe their execution memory model. A programming language's memory model is hardware-independent and also independently defines its own possible set of executions. Hence, there is the risk of mismatch between the programmer's view of a memory model in software and how the generated assembly code can execute on hardware. This mismatch is known to give rise to complex and subtle bugs [81, 82, 83, 84]. Consequently, it is critical to incorporate formal methods to describe and verify memory models.

Sequential consistency (SC) The strictest possible set of rules that can be enforced by a memory model is to maintain *sequential consistency* [85]. Sequential consistency enforces two properties:

- program order within a thread must be strictly obeyed; and,
- operations must be atomic and instantaneously visible to all threads.

These properties prohibit any reorderings of memory accesses, even if they are non-aliasing accesses, and disallow any optimisations that would mask the latency of writes.

Weak memory models SC is an expensive illusion to maintain, both in terms of hardware implementations and software optimisations. Instead of enforcing SC, most hardware architectures relax the SC properties *i.e.* they support *weak memory model*s. Weak memory models are less restrictive than SC in two ways:

- they permit particular memory reorderings; and,
- they may allow writes to propagate in different order to different processors.

Since these properties provide ample implementation freedom, there are a range of weak memory models.

In this section, we first discuss how hardware memory models support weak consistency (§2.3.1). Then, we discuss why programming languages adopted memory models (§2.3.2). Next, we discuss intricate details of the C11 memory model (§2.3.3). Finally, we present the C11 input sets and relations that we use in this thesis (§2.3.4).

2.3.1 Weak memory models originated from hardware optimisations

In pursuit of better memory parallelism and shorter access latencies, chip designers avoid adhering to SC requirements [32]. Instead, they support weaker memory guarantees and provide programmers with safety nets, in the event that stronger guarantees are required [86]. Most multi-processor architectures are weak since they allow 1) memory reorderings and 2) less-restrictive write propagation. Table 2.1 shows a list of multi-processor architectures collated by Adve *et al.* [32]. This list is not comprehensive, but provides the necessary intuition about weak memory models. This table shows the types of reorderings (§2.3.1.1) and safety nets (§2.3.1.2) of various multi-processors. We do not cover write propagation since the high-level synthesis of concurrent programs does not feature write buffers or caches. In the future, HLS of concurrent programs may incorporate write buffers or caches. In such any event, the atomicity of memory operations may no longer be guaranteed and hence there must exist a hardware mechanism via HLS to ensure write propagation when necessary.

Table 2.1: Categorisation of the various weak memory models [32].

WMM	$W \rightarrow R$	$W \rightarrow W$	$R \rightarrow R/W$	Safety Net
SC [85]				
IBM 370 [87]	✓			serialisation
TSO [88]	✓			RMW
PC [89]	✓			RMW
PSO [90]	✓	✓		RMW, STBAR
WO [91, 92]	✓	✓	✓	syncs
RC [89]	✓	✓	✓	rel-acq, RMW
Alpha [93]	✓	✓	✓	MB, WMB
RMO [94]	✓	✓	✓	MEMBARs
PowerPC [89, 95]	✓	✓	✓	SYNC

2.3.1.1 Relaxing memory ordering

They are three of types of reorderings that an architecture may permit: 1) reordering reads after writes to non-aliasing locations ($W \rightarrow R$); 2) reordering writes after writes to non-aliasing locations ($W \rightarrow W$); and 3) reordering reads or writes after reads to non-aliasing locations ($R \rightarrow R/W$). Different architectures allow different types of reorderings:

- **SC**: SC does not allow any memory reordering.
- **Reordering** $W \rightarrow R$: The IBM 370, Total Store Order (TSO) and Processor Consistency (PC) models allow reordering of reads after writes to non-aliasing locations. They are classified as total store order, since all writes must execute in-order.
- **Reordering** $W \rightarrow W$: The Partial Store Order (PSO) model also allows reordering of writes to non-aliasing locations and hence only partially orders stores.
- **Reordering** $R \rightarrow R/W$: The weak ordering (WO), release consistency (RC), Digital Alpha, SPARC V9's relaxed memory order (RMO) and IBM's PowerPC models allow all reorderings between memory accesses that do not alias.

2.3.1.2 Enforcing memory ordering via safety nets

In order to provide strong ordering guarantees for programmers, each architecture provides safety nets that enforce memory ordering:

- **SC**: SC does not require any safety nets since it does not allow any reorderings.
- **IBM 370**: IBM 370 requires special serialisation instructions to enforce ordering.
- **TSO and PC**: TSO and PC, on the other hand, do not provide any serialisation instructions but rather enforce order via read-modify-write instructions.
- **PSO and RMO**: Both PSO and RMO provide *STBAR* instruction that enforce

ordering between two writes.

- **RC**: In addition to RMWs, RC provide acquire-release pairs that are special synchronisation instructions that grant or relinquish access to a set of shared locations.
- **Alpha, RMO and PowerPC**: Alpha and RMO provide memory barriers (*MB*) to guarantee order between any memory operations. Alpha also provides *WMB* that only orders writes. PowerPC provides a *SYNC* instruction that is similar to *MB*.

2.3.2 Language standards began to adopt weak memory models

Chronologically, the first focus of weak memory models was on hardware architectures. Once researchers began to understand the issues with weak memory models on hardware, attention diverted to the fact the programming language standards were also inept in expressing multi-processor memory synchronisation correctly.

2.3.2.1 Java memory model

The first signs of problems were identified in the Java memory model (JMM) [96]. Careful studies of the JMM suggested that common compiler optimisations were violating the memory model rules in a multi-threaded environment. JMM only supported coherence and its support for volatiles was unclear. Additionally, the JMM allows prescient stores, where a store can be executed much earlier than the order specified within a thread. This optimisation consequently meant that the JMM could be broken with very simple and targeted tests [96]. The Java memory model was fixed by Manson *et al.* [97]. They provide a precise definition for thread interaction and well-formed executions with formal and informal semantics.

2.3.2.2 C/C++ memory concurrency

Soon after the well-studied issues of Java's memory model, the C/C++ pthread support was under scrutiny. For example, Boehm *et al.* [98] noticed that threads cannot be implemented and designed independently, similar to approaching each thread as a compilation of a C library. They show that compiling threads independently leads to several correctness issues during compilation and highlight three concrete examples [98, § 4]. They also highlighted that the C approach of multithreading was as under-specified as the Java memory model used to be.

Formalisation of C memory concurrency Several independent efforts to formalise the C11 took place simultaneously. Boehm *et al.*'s [99] approach is similar to the Java memory model [97], in that they present a data-race-free model for C/C++, which guarantees SC behaviour for programs without a data race and undefined behaviour for programs with a race. Batty *et al.* [100] build on the work of Boehm *et al.* and provide a mathematically rigorous semantics for C/C++ concurrency. This axiomatic model includes data-race freedom and supports reasoning for locks, fences and atomic operations. Both these works address a very important issue, one that is unique to C. These works support the definition of weakly consistent C atomics (they refer to them as low-level atomics). Supporting weak atomics is vital to reasoning about program correctness of performance-critical code. All of these these efforts were pivotal to the drafts and final C11 standard [101].

2.3.3 C11 memory model

The 2011 revision of the C language, 'C11', defines three types of memory operations: ordinary memory operations, atomic memory operations and memory fences [101, §5.1.2.4,§7.17]. Ordinary memory operations can either be regular loads or stores. Atomic operations can either be atomic loads, stores or read-modify-writes. A memory fence is a synchronisation operation that is not associated to a memory location. Atomic operations, *atomics*, are memory operations that: 1) must appear to be instantaneous 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 relax the properties of SC atomics, which are identified via consistency modes (also known as *memory order*). Finally, memory fences enforce ordering between operations before the fence and operations after the fence. Our benchmarks do not utilise fences, but we support them for correction purposes. Additionally, C11 also defines a set of weakly consistent fences.

In this section, we discuss the different C11 consistency modes (§2.3.3.1), how C11 defines runtime memory events and relations over these events to ensure legal execution (§2.3.3.3), an example that describes this process of converting a static C11 program into dynamic executions of memory events (§2.3.3.4) and, finally, discuss some related works that have emerged after the introduction of C11 (§2.3.3.5).

2.3.3.1 C11 consistency modes

Overall, the available C11 memory orders are: *SC*, *acquire*, *release*, *consume* and *relaxed.* The *consume* mode is not commonly used[3] and we treat *consume* as *acquire.* Each memory order applies its own set of rules. Firstly, different memory orders apply to particular types of memory operations:

- *SC* mode applies to atomic loads and stores;
- *acquire* mode applies only to atomic loads;

[3]In fact, it is temporarily deprecated. Source: `https://bit.ly/2Eb7j29`

- *release* mode applies only to atomic stores;
- *relaxed* mode applies to atomic loads and stores.

Secondly, each memory order disallows particular reorderings from occurring with respect to other memory operations within a thread:

- any *atomic* load or store cannot be reordered with any other atomic load or store that accesses the same location (this property ensures coherence);
- an *SC atomic* load or store cannot be reordered with any other load or store;
- an *acquire load* cannot be reordered with loads or stores that are ordered after it in program order;
- a *release atomic* store cannot be reordered with loads or stores that are ordered before it in program order; and
- a *relaxed atomic* places no additional restrictions on reorderings.

2.3.3.2 Read-modify-write operations

Atomic read-modify-write (RMW) operations [101] are special operations, since they are composite operations that combine two memory operations atomically: an atomic load and an atomic store. No other aliasing memory operations in the entire program are allowed to execute between this pair of operations. The C11 memory model express them as two individual operations, connected via a *rmw* relation (Event relations are discussed in the next section). This edge enforces the criterion that no aliasing memory accesses can occur between the load and store of the RMW. Consequently, a RMW's memory order applies to its load and store operations as if they were individual operations[4].

Atomic compare-and-swaps An atomic compare-and-swap (CAS) is a critical RMW operation, that allows more than two threads to synchronise with each other. A default CAS operation is expressed as a function with three arguments: an atomic location, an expected value and a desired value. If the atomic location holds the expected value, it is instantaneously swapped with the desired value, otherwise its value is unchanged. C11 also defines a set of weakly consistent CASes with two additional arguments: the memory order when the comparison succeeds and the memory order when the comparison fails. We only consider the the memory order when the comparison succeeds since it is always stronger.

Additionally, 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 [101, §7.17.7.4.4]. 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

[4]RMWs also support *acquire-release* mode, but we treat it as *SC*.

to how the CAS operation can be reordered with the other memory operations in its thread. We implement the strong CAS, as it is more powerful, and is required by our benchmarks.

2.3.3.3 Memory events and event relations

These C11 definitions of §2.3.3.1 are simple, easy to understand and sufficient to implement C11 correctly. These simplistic rules are implied from more complex and in-depth definitions of the C11, where its memory semantics is defined in terms of allowed *global executions* of a concurrent program. A concurrent C program consists of a set of memory operations that are executed dynamically at runtime. Since a program's execution is dynamic, there are an exponential number of possibilities that can be executed in runtime, defined as the set of *candidate executions.* C11 describes the set of allowed dynamic executions amongst the set of candidate executions, defined as the set of *legal executions*, C11 does so by defining a set of rules over memory events, *i.e.* event relations, to which all concurrent programs must adhere.

Firstly, a memory event is a runtime event, which executes a memory operation from the source program. An event has four parameters:

1. a read (R) or write (W);
2. a memory order of the memory event, where `SC`, `ACQ`, `REL`, `RLX` and `na` represent SC, acquire, release, relaxed and non-atomic accesses respectively;
3. a memory location that the memory event applies to;
4. and, finally, a exact value that is either read from or written at runtime.

A memory event requires a value parameter since different execution paths taken by a program can lead to different runtime values, especially for read events.

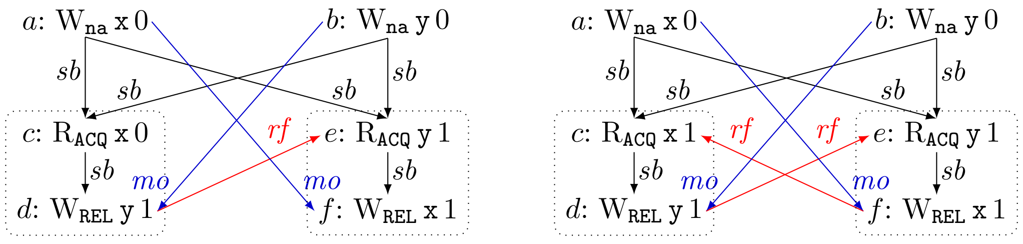
Additionally, C11 describes several binary relations over memory events. In contrast, the only binary relation that C11 defines over the set of static memory operations is *program order* (*po*). *po* is the intra-thread order of memory operations defined in the source program. The C11 memory event relations that are vital for this thesis are:

- *sequenced-before* (*sb*), relates two memory events that are related by *po*. The only difference is *po* relates memory operations whereas *sb* relates memory events.
- *reads-from* (*rf*), relates two memory events where one event reads the value written by another event to the same location. *rf* also enforces that the value written to and value read by must be the same. *rf* is a subset of *sw*.
- *modification-order* (*mo*), relates the order in which two write events to the same location are executed.

A candidate execution is legal if and only if we obtain an *acyclic* graph over particular binary relations of C11. A cyclic candidate execution can either be inconsistent or consists

```
                        atomic_int x=y=0;
r0=atomic_load_explicit(&x,    || r1=atomic_load_explicit(&y,
   memory_order_acquire);      ||    memory_order_acquire);
atomic_store_explicit(&y, 1,   || atomic_store_explicit(&x, 1,
   memory_order_release);      ||    memory_order_release);
                  assert(!(r0==1 && r1==1))
```

Figure 2.6: an example C11 program performing load-buffering.



(a) a legal execution

(b) an illegal execution

Figure 2.7: Two candidate executions of C11 program in Fig. 2.6.

a data race, whose full definitions can be obtained in Batty *et al.* [100, §2.10]. One of these consistency conditions is to ensure that the *happens-before* (*hb*) relation is acyclic. Loosely speaking, *hb* is the union of *sb* and *rf*. *hb* is also transitive in that if $(a, b) \in hb$ and $(b, c) \in hb$ then $(a, c) \in hb$. We visualise the importance of acyclicity by example.

2.3.3.4 An example

As an example of how this semantics works, consider the C11-style program in Figure 2.6. Here, we have a two-threaded program with two atomic variables, `x` and `y`. Both variables are initialised to zero, before spawning the two threads. The left thread atomically reads from `x`, and then atomically writes one to `y`, and the left thread atomically reads from `y` and then atomically writes one to `x`. Both reads are acquire loads and both writes are release stores. Weak atomic accesses in C11 are implemented using functions, where we explicitly state a location, a value (if it's a store) and, finally, a memory order. The final assertion states that we must not be able to read ones from both reads as that would mean that the writes were reordered with the reads in either thread, possibly because the loads are buffered (hence, the name of this litmus test: *load buffering* [100, §3]).

This example in Fig. 2.6 has sixteen candidate executions *i.e.* can be executed in sixteen possible ways during runtime. Fig. 2.7 show two executions of these executions: a legal execution on the left and an illegal execution on the right (all executions can be seen visually using `cppmem`[5], a tool developed by Sewell *et al.* [88] to verify and reason about

[5]An interactive online view can be obtained at `https://bit.ly/2BOOPpk`

C11 programs).

This program has six memory events (a,b,c,d,e,f), each with four parameters, that can be extracted from its source program. Two events, a and b, are from the initialisation step, and the other four events are distributed evenly between two threads by dotted rectangles.

The *sb* relation relates events that are executed in the order specified at source-code level *i.e.* when the static *po* relation is preserved during runtime by two memory events. Hence, we see that the initialisations a and b are sequenced before c, d, e and f. We also see that c is sequenced before d and that e is sequenced before f.

The *mo* relation relates the order in which two aliasing write events hit main memory. In both executions, we see that a writes to `x` before f and b writes to `y` before d.

The *rf* relation orders an event that reads a value written by another event. For example, e reads from d in both executions, since their value parameters are the same. However, in Fig. 2.7(b), we see that c also reads from f, instead of a, since the value read by f is one, leading to an illegal execution.

For this example, the union of *sb* and *rf* must not form a cycle to ensure that *hb* is acyclic. This is not true for the execution in Fig. 2.7(b), since (c, d), (d, e), (e, f) and (f, c) forms a cycle. This execution is inconsistent since we cannot identify the order in which c, d, e and f are executed, as they contradict each other. This cycle does not exist in Fig. 2.7(a), since c reads from a, instead of f. Hence, we can guarantee that c, d, e and f are executed in strict order.

2.3.3.5 C11 related works

Since the official release of the C11 standard, there have been many works related to the C11 memory model for various purposes.

Compilation verification to various architectures The key works involving C11 have been to implement it correctly within compiler infrastructures and various architectures. There are many works on compiling C/C++ correctly to specific architectures such as x86 [88], POWER [102] and ARM [86, 103]. The formalisation of C11 allowed several compiler issues to be identified [104]. CompCertTSO by Ševčík *et al.* [82] provides correctness guarantees that any compiler pass within the CompCert toolchain is correct for TSO-based memory models of x86. Morriset *et al.* [105] present several strategies to approach bug hunting for concurrency issues regarding to the C11 memory model. Vafeiadis *et al.* [81] identify compiler optimisations that misinterpret the C memory model and introduce subtle bugs.

Simplification and unification of C memory model On the theoretical side, several works to unify and simply the C memory model have been undertaken. Alglave *et al.* [83]

propose an axiomatic model for SC, TSO, C release-acquire semantics, Power and ARM and also offer a unifying simulation tool to allow concise specification of any of the above memory model. Batty *et al.* [106] also provide an in-depth discussion on how to simplify the semantics of SC atomics, which was rather convoluted and hard to understand prior to their work. As variation between memory models are becoming increasingly subtle, Wickerson *et al.* [107] present a technique to automatically compare memory models against each other and generate counter-examples.

Libraries of concurrent data structures Since weak atomics are notoriously hard to implement correctly, several works present a library of data structures that can be re-used by programmers. The idea is to avoid repeating the arduous reasoning process of memory consistency by providing a thoroughly verified set of data structures. Batty *et al.* [108] present a generalisation and abstraction for constructing complex concurrency libraries and use compositional reasoning for concurrent C programs. Norris and Demsky [109] present a collection of lock-free data structures that utilise weak atomics that are exhaustively explored for all possible weak behaviours.

GPUs also support C memory model OpenCL [110] also began supporting weak atomics with the inception OpenCL 2.0 [5, §3.3.4]. Alglave *et al.* [111] showed that GPUs are also susceptible to weak behaviours for specific litmus tests under certain conditions on an Nvidia GPU memory model. Within the GPU context, there is also the added complexity that different memory scopes exist, where different groups of threads have restricted access to certain memory hierarchies, as discussed by Wickerson *et al.* [84].

Operational models instead of axiomatic models Furthermore, some interesting work by Sarkar *et al.* [102] and Alglave *et al.* [83] on defining memory models operationally, rather than axiomatically, is also worth mentioning. These works highlight an important problem with an axiomatic model, which is that its execution state space grows exponentially with program size. Hence, simulation tools tend to verify axiomatic models up to a fixed number of memory events.

2.3.4 Inputs to our analyses

Having discussed the details of C11, all our analyses in this thesis require programs with the following inputs: the set of memory operations (§2.3.4.1) and input relations (§2.3.4.2).

2.3.4.1 Memory operation sets

Firstly, we describe the set of all static memory operations, V_{mem}, as follows:

$$V_{\text{mem}} = V_{\text{ld}} \cup V_{\text{st}} \cup V_{\text{fence}} \tag{2.2}$$

where V_{ld}, V_{st} and V_{fence} are the set of loads, stores and memory fences.

Then, we describe the set of all atomic operations, $V_{\text{at}} \subseteq V_{\text{mem}}$, as follows:

$$V_{\text{at}} = V_{\text{sc}} \cup V_{\text{acq}} \cup V_{\text{rel}} \cup V_{\text{rlx}} \tag{2.3}$$

where V_{sc}, V_{acq}, V_{rel} and V_{rlx} are the set of sequentially-consistent, acquire, release and relaxed atomics. The details of these different consistency guarantees were elaborated in §2.3.3.1. Additionally, acquire and release atomics only applies to loads and stores respectively *i.e.* $V_{\text{acq}} \subseteq V_{\text{ld}}$ and $V_{\text{rel}} \subseteq V_{\text{st}}$.

Furthermore, we describe set of all fences, V_{fence}, as follows:

$$V_{\text{fence}} = V_{\text{scfence}} \cup V_{\text{acqfence}} \cup V_{\text{relfence}} \tag{2.4}$$

where V_{scfence}, V_{acqfence} and V_{relfence} are the set of sequentially-consistent, acquire and release fences.

2.3.4.2 Input relations

Our analysis also relies on the following relations between memory 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, and finally
- *rmw*, the ‘read-modify-write’ relation, which relates a load and a store to the same location that are part of a read-modify-write operation.

2.4 Lock-free programming

Synthesising the C11 memory model provides the flexibility using atomics, which are fundamental synchronisation primitives that provide the means for safe concurrent access of shared memory without the presence of locks. Atomics are fundamentally different to locks, as atomics are non-blocking entities. An atomic access must execute instantaneously and cannot be interrupted. Atomic accesses are linearisable [7] *i.e.* appear to happen at discrete points in time without conflicts. In this section, we discuss the important classes of multi-threaded algorithms.

Blocking algorithms In contrast, lock-based data structures are generally implemented using *mutex*es. A suggested by its name, a mutex (*mut*ual *ex*clusion) guarantees exclusive access of shared data to a thread. Mutexes, however, are function calls that block until the lock is acquired or released respectively. As a result, lock-based concurrency are typically classified as blocking algorithms. Blocking algorithms halt the progress of threads that attempt to acquire a mutex, until they succeed. This scenario is bad because it prevents these threads from doing any useful work and possibly, in the worst case, causes them to to halt indefinitely because the lock is never released *i.e.* a *deadlock*. In addition to being susceptible to deadlocks, mutexes also serialise the shared memory accesses, which potentially worsens performance. Also, performance is typically reliant on usage granularity of these mutexes [34].

Non-blocking algorithms Concurrent C programs that use atomics, instead of locks, to synchronise across threads are *lock-free* programs. Lock-free programs can be written in a non-blocking manner. However, utilising atomics does not guarantee that a lock-free program is non-blocking. A *non-blocking* algorithm ensures that unexpected delays in one thread do not cause other threads to be delayed [7]. As a counter-example, one can imagine using atomics to implement a spinlock. Even though a spinlock does not halt a thread's execution at a function call since there is no mutex present, during runtime the execution is still stuck in the same region of code until the lock is acquired. In this regard, both blocking at a function call or spinning endlessly on an atomic variable are conceptually similar, since they both stop the thread from making progress. Hence, we are not interested in lock-free programs merely to avoid locks. We want to focus on lock-free programs that are also non-blocking.

Lock-freedom For the context of this thesis, a *lock-free* algorithm guarantees that the program as a whole is making progress *i.e.* at least one thread is making progress at any given point of time [7]. Lock-free algorithms do not block progress of competing threads. If two threads are competing to access the same lock-free data structure, one thread is guaranteed to succeed and the other thread is informed that it has failed. The failing thread has now the choice of trying again or performing other tasks *i.e.* the failing thread is not prevented from doing other useful things instead of halting. Wait-free algorithms provide stronger guarantees compared to lock-free algorithms. A *wait-free* algorithm guarantees that a thread will be able to make progress within a finite number of steps [7]. The downside of incorporating wait-free algorithms, compared to lock-free algorithms, is that the complexity and average time taken to complete a given task may be slower. We are not interested in wait-free programs in this thesis, but it is definitely a unique property that could be exploited in the future. Wait freedom can be particularly interesting since Alistarh *et al.* [112] show that a large class of lock-free algorithms are, in fact, wait-free under the scheduling conditions of commercially available hardware.

Benefits of lock freedom Empirically, lock-free programs are known to have good performance compared to their lock-based counterparts [113]. Due to their non-blocking nature and superior performance, a range of lock-free data structures are typically found as part of the Linux kernel [114] and high-performance libraries such as Boost [115] and libcds [116]. As lock-free programs are beginning to gain wider acceptance in the software world, we are motivated to enable lock-free C concurrency via the synthesis of the C11 memory model. Our work enables C programs with atomic operations to be directly synthesised and executed on reconfigurable hardware. Hence, we are enabling a plethora of lock-free data structures to be targeted to FPGAs for the first time.

2.5 Evaluation of our methods

Since HLS support of lock-free programs is relatively new, we evaluate our work on a set of hand-picked lock-free data structures from software benchmarks. In this section, we discuss the following:

- we introduce, describe and characterise our set of lock-free data structures, in §2.5.1;
- we define three data-flow patterns, which are based on different arrangements of our data structure routines, in §2.5.2;
- we introduce our memory- and compute-dominant experiments that we use to evaluate our work, in §2.5.3.2;
- and, finally, we describe the HLS, CAD and simulation tools and setups used throughout our work, in §2.5.4.

2.5.1 Benchmarks

We evaluate our method on three lock-free data structures: a single-producer-single-consumer buffer [117], the Treiber stack [118] and the Michael–Scott queue [119]. These data structures are real-world examples of lock-free data structures and are part of the Boost library [115] and Linux kernel [114]. We use weak versions of the stack and queue from Norris and Demsky [109].[6]

To access a lock-free data structure correctly in a concurrent setting, each data structure imposes a pre-defined ordered set of memory accesses, referred to as *routines*. Routines play an important role for all our experiments. Examples of data structure routines include pushing, popping, enqueueing and dequeueing. Routines provide an abstraction that enforces particular memory orderings when accessing these data structures, which ensures race-free and correct synchronisation between concurrent threads. Typically, the routines' the original implementation re-attempt their accesses until successful via while loops. We re-factor these routines to only execute a single attempt. We do so for two reasons. Primarily, re-attempts within these routines implicitly turn the stack and queue into blocking data structures, whereas we are interested in non-blocking data structures. Secondarily, avoiding `while` loops allows the entire routine to fit into a basic-block, which suited LegUp compilation.

In the remaining part of this subsection, we describe each data structure and how they support the specified synchronisation in a lock-free manner, before finally we characterise all their routines for comparison. All data structure routines are represented in Appendix A, with most memory accesses labelled for reference.

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

2.5.1.1 SPSC buffer

The SPSC buffer [117], as represented in Listing A.1 of page 176, only allows communication between two threads, a producer (only allowed to push) and a consumer (only allowed to pop). This buffer is first-in-first-out (FIFO), and therefore requires two atomic pointers, `tail` and `head`, protecting a non-atomic `array`. The `tail` points to the next location in `array` that the producer can push to while the `head` points the next element in `array` that the consumer must pop. Since it only allows two threads to synchronise, it can be implemented via atomic loads and stores without CASes.

Push and pop routines The producer checks if the buffer is not full (line 6) before pushing data to the `array` (line 9) and then updating the `tail` pointer (line 10). The consumer checks if the buffer is not empty (line 20) before popping data from the `array` (line 22) and then updating the `head` pointer (line 22). Both the pointer updates are done in a circular/modulo fashion via the `increment` function.

Routine synchronisations These push and pop routines ensure correct inter-thread memory behaviour by enforcing that the non-atomic `array` only synchronises as a consequence of its release-acquire pairs synchronising, *i.e.*

- ❸ synchronises with ❼, if the release store of ❹ and the acquire load of ❻ synchronise.
- ❼ synchronises with ❸, if the release store of ❽ and the acquire load of ❷ synchronise.

2.5.1.2 Treiber stack

A Treiber stack [118], as represented in Listing A.2 of page 177, allows communication between multiple threads. This stack is last-in-first-out (LIFO), and therefore requires a single atomic pointer, `top`, protecting a linked list that is implemented with two arrays, an atomic array `nodes_next` (holding all nodes' next pointers) and a non-atomic array `nodes_value` (holding all nodes' values). Since multiple threads can attempt to push and pop on the stack concurrently via a single atomic pointer, all accesses to update the `top` must be done via CASes.

Push and pop routines To push to the stack, first the top pointer is read (line 4). Then, a new node is allocated, where its next pointer is updated with the current top value (line 6). Finally, a CAS is performed to advanced the `top` pointer of the stack by one (line 8) to ensure the success of the push routine. To pop from the stack, the current `top` pointer is read (line 14) and the next pointer of the current node pointed to by `top` is read (line 15). Then, the stack is checked that it is not empty (line 16). If so, then the `top` pointer is advanced via a CAS operation (line 18). If the CAS succeeds, the pop routine is completed and the node value can be read (line 20).

Routine synchronisations The push and pop routines ensure correct inter-thread memory synchronisation by enforcing that the linked list's next pointer is always updated via atomic accesses (❸ and ❻) and this updates to the linked list are synchronised via atomic release-acquire pairs of the `top` pointer, *i.e.*

- ❸ synchronises with ❻, if the release store of ❹ and the acquire load of ❺ synchronise.
- ❻ synchronises with ❸, if the release store of ❽ and the acquire load of ❶ synchronise.

2.5.1.3 Michael–Scott queue

A Michael–Scott queue [119], as represented in Listing A.3 of page 178, allows communication between multiple threads. This queue is FIFO, and therefore requires two atomic pointers, `tail` and `head`, protecting a linked list that is implemented with two arrays, an atomic array `nodes_next` (holding all nodes' next pointers) and a non-atomic array `nodes_value` (holding all nodes' values). The `head` and `tail` point to the first and last nodes in the list respectively. The enqueue and dequeue routines of this queue concurrently update the linked list via these two atomic pointers. Since multiple threads can perform both enqueues and dequeues concurrently, all updates of `head` and `tail` must be executed via CASes.

Enqueue routine To enqueue a node to this queue, firstly the node must be allocated atomically (lines 6 to 9). Then, the `tail` value and the next pointer of the node pointed by `tail` are checked to be consistent (lines 12 to 15). If these values are consistent, then check if the `tail` is pointed to the last node, and if so, perform a CAS to enqueue the newly allocated node (lines 19 to 22). If the `tail` was not pointing to the last node, then it is updated (lines 26 to 28).

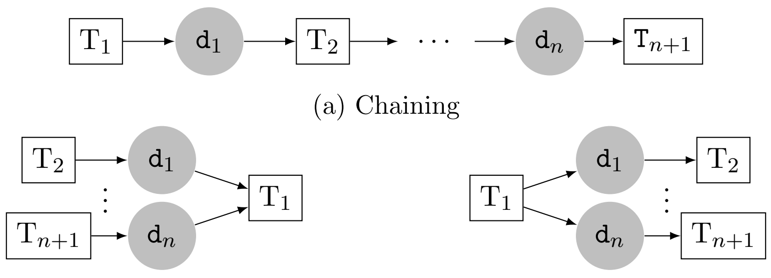
Dequeue routine To dequeue a node from this queue, firstly the `head` value and the next pointer to the node pointed by the `head` are checked to be consistent (lines 40 to 42). If these values are consistent and the queue is not empty, then the `head` value is updated to the next node in the list (lines 56 to 58) via a CAS operation. On the contrary, if the tail is failing behind, then the `tail` is updated (lines 47 to 51). Note that the dequeue routine accesses both atomic pointers whereas the enqueue only accesses the `tail`.

Routine synchronisations The enqueue and dequeue routines ensure correct inter-thread memory behaviour via the use of CASes and several release-acquire pairs that protect the linked list. These pairs are used to check the queue consistency at the start of each loop and also to update the next pointers, *i.e.*:

- ❶ can synchronise with ❻, ❼ or ⓬,
- ❽ can synchronise with ⓮, and
- ❹ can synchronise with ❷, ❺ or ❾.

Table 2.2: Our benchmarks and their memory access properties.

Benchmarks	push accesses		pop accesses		atomic accesses				
	total	atomic	total	atomic	acq	rel	rlx	na	CAS
SPSC buffer [117]	4	3	4	3	2	2	2	2	0
Treiber stack [118]	4	4	4	4	2	2	4	0	2
MS queue [119]	10	9	7	6	5	5	5	2	5



(a) Chaining

(b) Reduction

(c) Distribution

Figure 2.8: The experiments we conduct for each data structure. Squares represent threads, circles represent data structure objects and arrows represent data flow.

2.5.1.4 Memory characterisation

Table 2.2 characterises the memory access requirements of each data structure by routine and consistency modes, which provides several highlights. Firstly the ratio of atomic accesses to all memory accesses are high. In fact, the buffer and queue only has one non-atomic access per routine whereas the stack has none. Secondly, all data structures only use weak atomics, *i.e.* none of their weak version use any SC atomics. Thirdly, the stack and queue require CAS operations to synchronise across more than two threads. Finally, the queue has the most number of memory accesses with the most complex synchronisation.

2.5.2 Data-flow patterns

We evaluate our data structures on three common data-flow patterns. These three patterns are typically how data flows across threads: one-to-one (*chaining*), many-to-one (*reduction*) and one-to-many (*distribution*). Fig. 2.8 show these patterns visually. We test all three patterns on our data structures via their routines.

For each data pattern, we instantiate n independent data structure objects and $n + 1$ pthreads, where we scale n from 1 to 8. Within each thread, we run a fixed number of iterations (256). Each thread implements at least one data structure routine on at least one data structure object. The three data-flow patterns are set up as follows:

- *Chaining* as in Fig. 2.8(a): T_1 pushes to $\mathtt{d}_1$, T_i (for $2 \leq i \leq n$) pops data from $\mathtt{d}_{i-1}$ and pushes it to $\mathtt{d}_i$. We measure time taken by T_{n-1}.

- *Reduction* as in Fig. 2.8(b): 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. We measure time taken by T_1.
- *Distribution* as in Fig. 2.8(c): 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. We measure time taken by T_1.

2.5.3 Experiments

2.5.3.1 Memory-dominant experiments

The combination of three data structures and three data-flow patterns provide nine unique settings, which we refer to a set of *experiments*. We also refer to these experiments as a set of *memory-dominant* experiments, since its entire workload only comprises of data structure routines. Memory-dominant experiments provide good insights when considering relative performance of memory scheduling by our different analyses.

2.5.3.2 Compute-dominant experiments

Memory-dominant experiments do not include any computational workload, in conjunction to their memory workload. Thus, these experiments do not show the capabilities of our analyses in the presence of computation, especially when we evaluate loop pipelining. Hence, we introduce another set of experiments, the *compute-dominant* experiments. This set of experiments are designed to evaluate the effects of loop pipelining when computation is part of a thread's workload. We do so by including long-latency division operators within certain threads of our three data-flow patterns, as follows:

- *Chaining*: We divide all data popped from d_1 to d_{n-1} by three and push the result to d_2 to d_n within T_2 to T_n respectivelym where we increment data pushed by T_1 by 3^n.
- *Reduction*: We divide all data popped from d_1 to d_n by three within T_1. We also increment the data pushed by T_2 to T_{n+1} by three.
- *Distribution*: We divide all data by three before pushing them to d_1 to d_n within T_1. We also increment all data by three, also within T_1, before the division.

2.5.4 Experimental Setup

We utilise LegUp 5.1 to synthesise each pthread as a hardware accelerator, with shared memory implemented on the FPGA. The memory architecture generated by LegUp when using their pthread flow was discussed in §2.2.5.3. We perform cycle-accurate simulation on our all experiments using vsim, which is invoked by LegUp. We place-and-route all our designs using Quartus 15.0 for a Cyclone V SoC FPGA (5CSEMA5) with 32075 ALMs, 128300 registers, and 3970 Kb of RAM blocks. We extract the clock frequency and resource utilisation data of our designs from Quartus' post place-and-route reports.

3. Scheduling Fine-grained C Concurrency for High-Level Synthesis

3.1 Introduction

When writing multi-threaded programs for conventional multi-processors, the most efficient way of synchronising threads is to use fine-grained *atomic operations* ('atomics') – as opposed to, for instance, coarse-grained mutual exclusion based on locks [113]. Despite the benefits of fine-grained concurrency in the form of atomics, state-of-the-art *high-level synthesis* (HLS) tools do not sufficiently support them [8, 120]. One approach for implementing atomics is to wrap each atomic operation in its own critical section by using pthread mutexes [8]. Each atomic operation is surrounding by a pair of **lock()** and **unlock()** function calls, which implicitly enforces mutual exclusion. Although this approach is correct, there are three problems with such an approach.

Firstly, atomics are the fundamental primitive of memory synchronisation. More complex synchronisation primitives such as test-and-set, spinlocks and semaphores are built based on the guarantees of atomic operations [7]. Hence, the HLS approach of implementing atomics using locks is problematic because it is a counter-intuitive starting point for the software community, who are the prospective HLS audience. Secondly, the usage of locks to implement atomics re-introduces memory serialisation and deadlocks, which contradicts the goal of implementing lock-free memory synchronisation. Thirdly, a lock's function calls also introduce performance overheads and inhibit loop pipelining.

To enable efficient high-level synthesis of lock-free programs, we focus on the two properties of an atomic operation. Firstly, an atomic operation must be an indivisible memory operation, which is guaranteed by current HLS compilation procedure for all memory operations. Secondly, an atomic operation must also obey specific memory orderings as per the C memory model, discussed thoroughly in §2.3.3. To satisfy these ordering guarantees, we frame the implementation of atomics as a HLS *scheduling* problem, where memory orderings can be represented using HLS scheduling constraints.

Since the C memory model only requires atomics to preserve memory orderings within its own thread, we can express these memory orderings as *intra-thread* memory constraints. These intra-thread memory constraints are then applied during the memory scheduling of individual threads. Although intra-thread constraints only affect the schedule of their respective threads, the C memory model guarantees that, when these individual thread schedules interact with each other, correct global memory behaviour is preserved.

In summary, we can implement atomic accesses by imposing additional intra-thread memory constraints on them. We inject these additional constraints based on certain scheduling rules, whereby these rules collectively form a *memory model.* Since our work poses the implementation of atomics as a scheduling problem, we are able to devise the first HLS method capable of compiling weak atomics, which we verify via automated model checking. We do so by using Alloy [121] to ensure our generated hardware is correct-by-construction. In this chapter, we discuss the following:

- In §3.2, we formalise the memory model generated by state-of-the-art HLS schedulers of multi-threaded programs by defining their intra-thread memory dependencies;
- In §3.3, we show why this memory model cannot synthesise any form of memory synchronisation across threads without relying on locks;
- In §3.4, we introduce a running example to visually capture the effects of our work on scheduling outcomes, where this example will be re-use in subsequent chapters to compare memory scheduling of various analyses.
- In §3.5.2, we augment the HLS schedulers to impose additional *intra-thread* dependencies on atomic accesses, generating a memory model that can support SC atomics;
- In §3.5.3, we further modify the HLS schedulers to tailor these additional dependencies of atomics based on their consistency guarantees, generating a memory model that can support *weak* atomics;
- In §3.6, we verify via automated model checking that our memory models generate hardware that implement atomics correctly, as per the C11 standard;
- In §3.7, we implement our methods in the LegUp 5.1 HLS tool;
- In §3.8, we discuss how we implement CASes in LegUp, which is required to synchronise more than two threads;
- Finally, in §3.9, we evaluate our method on a set of experiments, .

3.2 Formalising the state-of-the-art

Throughout our work, we focus on the *scheduling* stage of synthesis in which software instructions are assigned to hardware clock cycles[1]. Typical, HLS schedulers seek to maximise instruction-level parallelism by allowing independent instructions to be executed out-of-order. In particular, typically, non-aliasing memory accesses, or those that exhibit

[1]We assume that operations of all threads are executed within the same clock domain.

only read-after-read dependencies (*e.g.* `x=z; y=z`), can be reordered. These reorderings are invisible in a single-threaded context, but they can introduce unexpected behaviours in a multi-threaded context. For instance, if another thread is simultaneously writing to `z`, then reordering two instructions above may introduce the behaviour where `x` is assigned the latest value but `y` gets an old one.

The implication of this is not that existing HLS tools are wrong; these optimisations can only introduce new behaviours when the code already exhibits a race condition, and races are deemed a programming error in C [101, §5.1.2.4]. Rather, the implication is that if these memory accesses are upgraded to become atomic (and hence allowed to race), then existing scheduling constraints are insufficient. Hence, in this section, we first summarise the existing HLS memory dependencies injected during the scheduling stage of a *sequential* C program. Then, we follow this discussion with how these memory dependencies are naively extended to *concurrent* C programs and are insufficient to support fine-grained memory synchronisation across threads without locks.

3.2.1 Memory dependencies for sequential programs

Our formalisation refers to the following contents from our Background chapter:

- the input memory operation sets and memory relations, given in §2.3.4 of page 52;
- and, the SDC formulation, described thoroughly in §2.2.6.2 of page 40.

Also, we assume the absence of loop pipelining in this chapter and Chapter 4. Hence, all dependence distance are zero for all constraints of both chapters. Loop pipelining is discussed thoroughly in Chapter 5.

Memory dependencies (*mem*), which hold between memory operations, $V_{\mathrm{mem}} \subseteq V_{\mathrm{op}}$, are a subset of data dependencies ($mem \subseteq dd$). C-based HLS tools perform alias analysis on a sequential C program and preserve read-after-write (RAW), write-after-write (WAW) and write-after-read (WAR) dependencies between aliasing memory operations, which can be formally described as follows:

$$\begin{aligned} \mathit{mem\text{-}alias} = \{(v, v', 0) \mid (v, v') \in \mathit{po} \wedge (v, v') \in \mathit{sloc} \wedge \\ (v \in V_{\mathrm{st}} \vee v' \in V_{\mathrm{st}})\} \end{aligned} \tag{3.1}$$

where $V_{\mathrm{st}} \subseteq V_{\mathrm{mem}}$ is the set of store operations, *po* is the 'program order' relation and *sloc* is the 'same location'. *mem-alias* expresses that for every memory operation v that is ordered before any memory operation v', where both accesses are to the same location and either one is a store, there must be exist an ordering edge between v and v'.

The memory model enforced by state-of-the-art HLS tools on sequential C programs, *mem*, is defined as follows:

$$mem = \mathit{mem\text{-}alias} \cup \mathit{nopipe} \tag{3.2}$$

where *nopipe* inhibits any overlap between successive iterations:

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

nopipe enforces that all memory operations within an iteration (v) must be executed before any memory operations (v') from the next iteration, via a dependence distance of one, which prohibits any opportunities for loop pipelining.

In summary, *mem* only preserves memory ordering between aliasing memory operations where at least one of them is a store. In other words, this memory model omits orderings between all non-aliasing memory operations and also between operations that have RAR dependencies. Although these omissions afford better memory parallelism of sequential C programs, such optimisations are only *legal* in a single-thread context.

3.2.2 Synthesising multi-threaded programs

Unfortunately, *mem* is also applied individually to each thread of a concurrent C program. This is because memory scheduling of individual threads is treated as memory scheduling of a collection of sequential C programs. HLS tools enforce *mem* within each thread and then connect all threads to their shared memory constructs via hardware interfacing and arbitration, as discussed in §2.2.5.3 of page 37. These intra-thread constraints are, thereby, too weak to support any form of memory synchronisation between threads without locks.

3.3 Demonstrating the gap in current HLS capabilities

To understand why the current HLS memory model cannot support atomic operations, we provide two simple multi-threaded programs: "coherence" and "message-passing". These are two examples designed to test whether a memory model confronts to *sequential consistency* (SC), where all memory accesses appear to occur instantaneously and in the same order as the corresponding instructions in each thread [85].

These two examples can demonstrate unexpected behaviours, when *mem* of (3.2) from page 64 is applied individually to each thread. In both cases, the unexpected behaviour only arises when particular instruction sequences are carefully contrived, but we argue that similar sequences could easily occur in 'realistic' programs too. To make our examples concrete, we present actual LegUp schedules demonstrating these behaviours occurring. In these schedules, all memory locations are identified to be non-aliasing and are instantiated

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

Figure 3.1: A minimal example of a coherence violation. The assertion failing indicates a coherence violation, where the second load of `x` (`r1`) happens before the first load of `x` (`r0`).

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

Figure 3.2: A program that can exhibit a coherence violation when compiled using LegUp, if atomics are treated as non-atomics (thread T1 is launched with $\texttt{a} = 1$).

as LegUp global memory, where a load takes two cycles and a store takes one cycle [41].

We emphasise that the unexpected behaviours discussed here do not mean that LegUp's scheduler is wrong, because LegUp does not claim to provide support for C11 atomics. Rather, we use these examples to demonstrate how the scheduling rules need to be altered to handle atomics correctly, and thereby avoid the problematic cases demonstrated in this section. Although our examples are based on the LegUp HLS tool, they are relevant to any HLS tool that performs constraint-based scheduling on a per-thread basis.

Coherence One of the simplest violations of SC is a *coherence* violation [122, §8], as illustrated in Fig. 3.1. The atomic variable `x`, initially zero, is shared between two pthreads, `T1` and `T2`, that are synthesised as parallel hardware units executing concurrently. A coherence violation occurs when the first load (line 1.2) observes `x`'s new value but the second load (line 1.3) observes `x`'s old value[2]. This violation is detectable via the final-state assertion. This violation of coherence is due to the absence of read-after-read (RAR) edges in *mem-alias* of (3.1) from page 63.

We can observe a coherence violation in LegUp by first making some innocuous transformations to the source code, as shown in Fig. 3.2. The key objective in Fig. 3.2 is to increase the priority of the second load of `x` compared to the first one in an ASAP

[2]By default, all atomic operations are SC unless a memory order is specified.

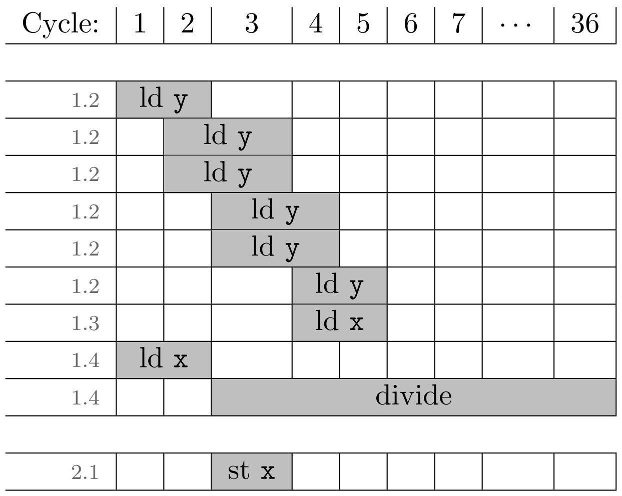


Figure 3.3: Schedules for threads T1 (top) and T2 (bottom) that allow the program in (b) to exhibit a coherence violation.

```
            int x=0; atomic_int y=0;
   T1() {                   ||    T2() {
1.1  x=1;                   || 2.1  int r0=0,r1=0;
1.2  atomic_store(&y,1);    || 2.2  r0=atomic_load(&y);
   }                        || 2.3  if(r0==1) r1=x;
                            ||    }
```

assert(r0 $= 1 \Rightarrow$ r1 $= 1$)

Figure 3.4: A minimal example of a message-passing violation. The assertion failing indicates a message-passing violation, where the flag is set (r0 $= 1$) but the data is stale (r1 $= 1$).

scheduling context. We do so with two steps. First, we replace the atomic variables with their volatile counterparts, to simulate atomics as unoptimised regular loads and stores. Then, we chain the second load's value into a division (whose denominator is set to 1 at runtime) and also inject extra loads of y that are part of an addition chain.

These transformations result in the schedule shown in Fig. 3.3.[3] Because of the division's large latency, the scheduler seeks to schedule the second read of x as early as possible. It determines that line 1.4 does not depend on line 1.3 (there is only a read-after-read (RAR) dependency on x) or on line 1.2, and hence can be executed on the first cycle. The repeated reads of y cause a delay between the two reads of x, and it is during this gap (third cycle) that thread T2 updates x. T2 only update x at the third cycle because LegUp takes two cycles to spawn consecutive threads. The resultant execution shows that the second load of x reads stale data because the first load of x reads the latest value.

[3]The schedule is constrained by dual-ported shared memory access, since LegUp global memory is implemented using block RAMs.

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

Figure 3.5: A program that can exhibit a message-passing violation when compiled using LegUp, if atomics are treated as non-atomics (thread `T1` is launched with `a` $= 3$).

Cycle:	1	2	3	4	5	···	35	36
1.2	ld `a`							
1.2			divide					
1.2								st `x`
1.3	st `y`							
2.1			ld `y`					
2.2			ld `x`					
2.2					slt `y==1? x:null`			

Figure 3.6: Schedules for threads `T1` (top) and `T2` (bottom) that allow the program in (b) to exhibit a message violation.

Message-passing Another example of an SC violation is illustrated by a failure of the *message-passing* paradigm [122, §3], which is illustrated in Fig. 3.4. This example involves two shared locations, `x` and `y`, where `x` represents a message being passed from thread `T1` to thread `T2`, and `y` is used as a 'ready' signal. A message-passing violation occurs if `T2` observes that `y` is set (line 2.3) but observes that `x` is still 0, as enforced by the final-state assertion. The reason for this violation is that there are no aliasing memory operations to preserve within both threads.

As before, some innocuous code transformations can coax the ASAP scheduler into revealing this behaviour, as shown in Fig. 3.5. This time, our objective is to delay the store of `x` (line 1.1) so that it executes after the atomic store of `y` (line 1.2). This scenario is easy to achieve. We simply delay the store of `x` by computing its store value via a division (line 1.1).

Fig. 3.6 shows the resultant schedule, where this high-latency operation delays the store to `x`. Because lines 1.1 and 1.2 are deemed independent, the scheduler permits them to be reordered and the store of `y` can be executed on the first cycle. In the reading thread (`T2`), both loads are scheduled simultaneously having used if-conversion [123] to replace

Cycle:	1
`a=42;`	$\text{st}_{\texttt{na}}$ `a` 42
`ast(&b,1,REL);`	$\text{st}_{\texttt{REL}}$ `b` 1
`x=17;`	$\text{st}_{\texttt{na}}$ `x` 17
`ast(&y,1,REL);`	$\text{st}_{\texttt{REL}}$ `y` 1

Figure 3.7: A running example of four non-aliasing stores, of which two are store releases, and its schedule based on *mem* in (3.2) from page 64.

the control flow with predicated statements via a select (or `slt`) statement in line 2.2. Again, since this thread is spawned by LegUp after two cycles, we can observe the new value of `y` but the old value of `x` – a violation of message passing.

In summary, we show the current HLS memory models are too weak and fragile to support atomic operations. Even innocuous code transformations can affect the scheduling priorities of memory operations and tool-specific details, such as the way LegUp spawns its threads, can influence the outcomes of memory synchronisation. Hence, in §3.5, we enforce additional scheduling rules on atomics to generate memory models that can support the execution of lock-free synchronisation.

3.4 A running example

Before we delve into our methods for supporting atomics, we present a running example that helps visualise the scheduling outcomes of the various memory models we propose. The example will also be revisited in subsequent chapters for comparison purposes.

In Fig. 3.7, we propose an example with a single thread that stores to four different memory locations: two of which are atomic locations (`b` and `y`) and release stores (`REL`). Each shaded cell represents the cycle in which the particular memory event is scheduled (memory events were discussed in §2.3.3.3). We also assume that each store takes one cycle and we assume ASAP scheduling with an unconstrained number of memory ports.

The schedule in Fig. 3.7 shows our running example implemented using *mem* in (3.2) from page 64. The scheduler treats atomics as ordinary operations, and since these memory accesses do not alias, all four memory operations are free to be scheduled simultaneously. Hence the total latency of this example is just one cycle. However, in presence of atomics, this schedule can be wrong since the store releases are scheduled earlier than they should be. We show how to alleviate this problem in §3.5.

3.5 Injecting HLS ordering constraints to support atomics

Thus far, we have seen how the current HLS memory models, via *mem* of (3.2) from page 64, are too weak to support atomics. In this section, we systematically enlarge *mem* to allow correct execution of atomics via additional scheduling rules. These rules are generic for all programs, but generate program-specific *intra-thread* scheduling constraints for each thread of an input concurrent C program. These constraints are then applied during memory scheduling of individual threads.

Our method supports C atomic loads, stores and read-modify-writes as well as fences. The input memory operation sets and memory relations required by our method are introduced in §2.3.4 from page 52. In this section, we discuss four memory models that enlarge the *mem* relation of (3.2) from page 64:

- In §3.5.1, we introduce a memory model that implements SC [85]. This implementation is straightforward but forbids any memory parallelism.
- In §3.5.2, we introduce a memory model that can support SC atomics. This memory model treats all atomics as SC atomics, which is the default consistency mode when left unspecified.
- In §3.5.3, we introduce a memory model that can support weak atomics. This memory model is sensitive to each atomic's consistency mode.
- Finally, in §3.5.4, we introduce a memory model that is extended to support memory fences.

3.5.1 Preserving SC semantics

A naive solution to ensure correct memory behaviour is to serialise all memory operations, regardless of any alias analysis or any atomics. This is achieved by defining *sc* as follows:

$$sc = \{(v, v', 0) \in V_{\text{mem}} \times V_{\text{mem}} \mid (v, v') \in po\}. \tag{3.3}$$

sc enforces strict order between all every pair of memory operations (v, v'), where v is ordered before v'. This memory model entirely overrules *mem*. Consequently, *sc* disallows any memory parallelism since it will only allow one memory operation to be executed in any given cycle.

Running example Fig. 3.8 shows the schedule of our running example based on *sc*. *sc* serialises all four memory accesses, resulting in a latency of four cycles, even though all accesses are non-aliasing and some accesses are non-atomic.

Cycle:	1	2	3	4
`a=42;`	st$_{\texttt{na}}$ `a` 42			
`st(&b,1,REL);`		st$_{\texttt{REL}}$ `b` 1		
`x=17;`			st$_{\texttt{na}}$ `x` 17	
`st(&y,1,REL);`				st$_{\texttt{REL}}$ `y` 1

Figure 3.8: A running example of four non-aliasing stores, of which two are store releases, and its schedule of based on *sc* from (3.3) of page 69.

Cycle:	1	2	3	4
`a=42;`	st$_{\texttt{na}}$ `a` 42			
`ast(&b,1,REL);`		st$_{\texttt{REL}}$ `b` 1		
`x=17;`			st$_{\texttt{na}}$ `x` 17	
`ast(&y,1,REL);`				st$_{\texttt{REL}}$ `y` 1

Figure 3.9: A running example of four non-aliasing stores, of which two are store releases, and its schedule of based on *mem-sc* from (3.4) of page 70.

3.5.2 Exploring SC atomics

Now, we now define a memory model that can support SC atomics. We introduce two additional rules that generates intra-thread memory dependencies for each *atomic operation* within each thread. $V_{\text{at}} \subseteq V_{\text{mem}}$ is the set of all atomics where we treat them all as SC atomics. This implementation is conservative but simple to understand and easy to implement. The two additional rules we introduce, $at\uparrow$ and $at\downarrow$, prevent atomics from moving 'earlier' or 'later' in the schedule respectively:

$$mem\text{-}sc = mem \cup at\downarrow \cup at\uparrow \tag{3.4}$$

where

$$\begin{aligned} at\downarrow &= \{(v, v', 0) \mid (v, v') \in po \wedge v \in V_{\text{at}}\} \\ at\uparrow &= \{(v, v', 0) \mid (v, v') \in po \wedge v' \in V_{\text{at}}\}. \end{aligned}$$

$at\downarrow$ specifies that for every atomic operation v that is ordered before any memory operation v', there must exist an ordering edge from v to v'. $at\uparrow$ specifies that for every memory operation v' that is ordered after any atomic operation v, there must exist an ordering edge from v to v'. The combination of these two constraints and *mem* sufficiently implements a memory model that can support SC atomics.

Running example Fig. 3.9 is the schedule of our running example when we implement *mem-sc*. The atomic store of `b` is constrained to execute after the store of `a` and before the store of `x` and `y` (by $at\uparrow$ and $at\downarrow$ respectively). The atomic store of `y` is constrained to execute after the stores of `a`, `b` and `x` (by $at\uparrow$). Hence, the latency of this example is four cycles. Notice that the resultant schedule is the same as when we apply *sc* of (3.3)

Cycle:	1	2	3	4
`a=42;`	$\mathrm{st}_{\texttt{na}}$ `a` 42			
`ast(&b,1,REL);`		$\mathrm{st}_{\texttt{REL}}$ `b` 1		
`w=5;`			$\mathrm{st}_{\texttt{na}}$ `w` 5	
`x=17;`			$\mathrm{st}_{\texttt{na}}$ `x` 17	
`ast(&y,1,REL);`				$\mathrm{st}_{\texttt{REL}}$ `y` 1

Figure 3.10: A slightly modified running example with five non-aliasing stores instead of four, of which two are store releases, and its schedule of based on *mem-sc* from (3.4) of page 70.

from page 69, as shown in Fig. 3.8. It turns out that *mem-sc* can produce schedules as conservative as *sc*. This can happen when $|V_{\text{at}}| \approx |V_{\text{mem}}|$ or when atomic operations appear in alternation to non-atomic operations, as in this example.

The difference between *mem-sc* and *sc* can be shown by adding another non-atomic store to a new location (let's say `w`) in middle of the four stores in our running example. Fig. 3.10 shows the schedule when applying *mem-sc* to these five memory accesses. The store of `w` must execute after the store of `b` and before the store of `y` (by $at\ddagger$ and $at\dagger$ respectively). However, the store of `w` and `x` are unordered and hence they can be executed in parallel. In contrast, *sc* enforces memory serialisation, which means it will produce a latency of five cycles with no memory parallelism for this modified example.

3.5.3 Exploiting weak atomics

Next, we define a memory model that can support weak atomics by making our scheduling rules sensitive to the consistency mode of atomic accesses. Recall, from §2.3.4, that V_{sc}, V_{acq}, V_{rel}, and V_{rlx} are the sets of sequentially consistent, acquire, release and relaxed atomics, such that $V_{\text{sc}} \cup V_{\text{acq}} \cup V_{\text{rel}} \cup V_{\text{rlx}} = V_{\text{at}}$. We require five additional rules to support atomics: two for SC atomics, one for acquire atomics, one for release atomics and one to preserve RAR dependencies for all atomics. *mem-weak* is a memory model that supports weak atomics, as follows:

$$\begin{aligned} \mathit{mem\text{-}weak} = \mathit{mem} \cup \mathit{sc}\ddagger \cup \mathit{sc}\dagger \cup \\ \mathit{acq}\ddagger \cup \mathit{rel}\dagger \cup \mathit{rar} \end{aligned} \tag{3.5}$$

Cycle:	1	2	3
`a=42;`	st$_{\texttt{na}}$ `a` 42		
`st(&b,1,REL);`		st$_{\texttt{REL}}$ `b` 1	
`x=17;`	st$_{\texttt{na}}$ `x` 17		
`st(&y,1,REL);`			st$_{\texttt{REL}}$ `y` 1

Figure 3.11: A running example of four non-aliasing stores, of which two are store releases, and its schedule of based on *mem-weak* from (3.5) of page 71.

where

$$\begin{aligned} \mathit{sc}{\downarrow} &= \{(v, v', 0) \mid (v, v') \in \mathit{po} \wedge v \in V_{\mathrm{sc}}\} \\ \mathit{sc}{\uparrow} &= \{(v, v', 0) \mid (v, v') \in \mathit{po} \wedge v' \in V_{\mathrm{sc}}\} \\ \mathit{acq}{\downarrow} &= \{(v, v', 0) \mid (v, v') \in \mathit{po} \wedge v \in V_{\mathrm{acq}}\} \\ \mathit{rel}{\uparrow} &= \{(v, v', 0) \mid (v, v') \in \mathit{po} \wedge v' \in V_{\mathrm{rel}}\} \\ \mathit{rar} &= \{(v, v', 0) \mid (v, v') \in \mathit{po} \wedge \mathit{sloc}(v, v') \wedge \\ &\qquad v \in V_{\mathrm{at}} \cap V_{\mathrm{ld}} \wedge v' \in V_{\mathrm{at}} \cap V_{\mathrm{ld}}\}. \end{aligned}$$

$\mathit{sc}{\uparrow}$ and $\mathit{sc}{\downarrow}$ define the ordering dependencies for SC atomics, which are similar to $\mathit{at}{\uparrow}$ and $\mathit{at}{\downarrow}$ in *mem-sc* of (3.4) from page 70, except that they only apply to SC atomics rather than all atomics. $\mathit{acq}{\downarrow}$ defines that acquire atomics cannot move 'down' in the schedule: for every acquire atomic v that is ordered before any memory operation v', there must exist an ordering edge from v to v'. $\mathit{rel}{\uparrow}$ defines that release atomics cannot move 'up' in the schedule: for every release atomic v' that is ordered after any memory operation v, there must exist an ordering edge from v to v'. *rar* defines that read-after-read (RAR) dependencies must be enforced for all atomics: for every atomic load v that is ordered before any atomic load v' to the same location, there must exist an ordering edge from v to v'. This rule differentiates relaxed atomics from non-atomic memory operations since non-atomic memory operations do not enforce RAR dependencies.

Running example Fig. 3.11 is the schedule of our running example when we implement *mem-weak*. $\mathit{rel}{\uparrow}$ enforces that store of `b` must execute after the store of `a` and that the store of `y` must execute after all memory accesses. However, the store of `x` is non-atomic and not constrained by *mem-weak*. Hence, in an ASAP schedule, this store of `x` can be reordered with the stores of `a` and `b` and executed in the first cycle. By supporting weak atomics, we can achieve a latency of three cycles, instead of four by *mem-sc* in Fig. 3.8.

3.5.4 Supporting memory fences

The C memory model also defines the behaviour of memory fences [101]. Under our assumption that the indivisibility of our memory accesses are guaranteed, memory fences also can also be implemented via scheduling constraints, similar to atomic operations.

In this section, we discuss the scheduling constraints necessary to support C11 fences. Although none of our benchmarks utilise fences, we include fences for correctness purposes.

Recall, from §2.3.4, that V_{scfence}, V_{acqfence} and V_{relfence} be the set of sequentially-consistent, acquire and release fences. We support memory fences with three additional rules: one each of SC, acquire and release fences. We implement a memory model, *mem-weak-fence*, that supports weakly consistent atomics and memory fences as follows:

$$\textit{mem-weak-fence} = \textit{mem-weak} \cup \textit{sc-fence} \cup \textit{acq-fence} \cup \textit{rel-fence} \tag{3.6}$$

where

$$\begin{aligned}
\textit{sc-fence} &= \{(v, v'', 0) \mid (v, v') \in \textit{po} \wedge (v', v'') \in \textit{po} \wedge v' \in V_{\text{scfence}}\} \\
\textit{acq-fence} &= \{(v, v'', 0) \mid (v, v') \in \textit{po} \wedge (v', v'') \in \textit{po} \wedge v' \in V_{\text{acqfence}} \wedge v \in V_{\text{ld}}\} \\
\textit{rel-fence} &= \{(v, v'', 0) \mid (v, v') \in \textit{po} \wedge (v', v'') \in \textit{po} \wedge v' \in V_{\text{relfence}} \wedge v'' \in V_{\text{st}}\}.
\end{aligned}$$

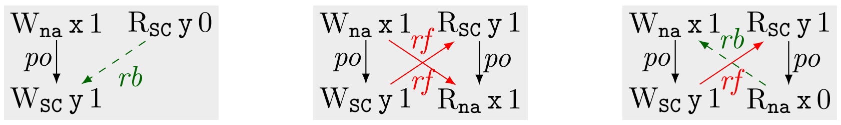
sc-fence defines that for each SC fence operation v', there must exist an ordering edge between v and v'', such that v is a memory operation ordered before memory fence v' and v'' is a memory operation ordered after memory fence v'. *acq-fence* defines that for each acquire fence operation v', there must exist an ordering edge between v and v'', such that v is a load operation ordered before memory fence v' and v'' is a memory operation ordered after memory fence v'. *rel-fence* enforces that for each release fence operation v', there must exist an ordering edge between v and v'', such that v is a memory operation ordered before memory fence v' and v'' is a store operation ordered after memory fence v'. Intuitively, fences can be thought of as upgrading some memory operations to be atomic operations. *acq-fence* upgrades all loads ordered before the memory fence into acquire loads, *rel-fence* upgrades all stores ordered after the memory fence into release stores and finally *sc-fence* combined the effects of *acq-fence* and *rel-fence* for SC fences.

3.6 Ensuring correctness via automated model checking

Since the C11 memory model is complex, it is critical that our additional rules are verified formally to ensure correct behaviour on our generated hardware. Even though our scheduling constraints are relatively straightforward to realise, it is still challenging to justify that our rules are sufficient to eliminate all executions that are deemed inconsistent by the C11.

3.6.1 Verifying our method

The C standard does not define the meaning of individual atomic accesses, but rather in terms of which executions of an entire program are allowed and which are not, as we described in §2.3.3. Let us consider three candidate executions of our message-passing example in Fig. 3.4, each of which consists of memory events (discussed in §2.3.3.3)



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.* [124].

Let us define a *buggy execution* to be an execution that is forbidden by C yet allowed by our method. The existence of such an execution would demonstrate that our method does not preserve enough of 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 of our implementation is more subtle.

As discussed earlier, the only source of memory reorderings in our implementation is instruction reorderings. Therefore, our starting point is to characterise the executions allowed by SC. Shasha and Snir [125] characterise SC executions using the rule

$$\mathbf{acyclic}(po \cup rf \cup mo \cup rb) \tag{3.7}$$

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 in (3.7) by replacing *po* with the ordering constraints generated by our scheduling rules of *mem-weak-fence* in (3.6) from page 73:

$$\mathbf{acyclic}(mem\text{-}weak\text{-}fence \cup rf \cup mo \cup rb).$$

This rule has the same effect as (3.7) applied to a less constrained program order.

3.6.2 Modelling our local analysis in Alloy

To build confidence that our work implements C11 atomics correctly, we use the Alloy model checker [121] throughout our development process. Previous work has *proved* the correctness of C11 implementations both on CPUs [100] and on GPUs [84], but such proofs are laborious and fragile, and hence ill-suited to our prototype implementation. Therefore, we turn to *lightweight* methods for verifying correctness via Alloy. Alloy is a mature and well-supported tool whose input language (based on relational algebra) closely matches the style in which most memory models are written. It has previously been used by Wickerson *et al.* [107] to check C11 and OpenCL memory model implementations for several CPU and GPU architectures. Here, we port their work to HLS.

Alloy refers to both its analyser and its programming language. The Alloy analyser translates models into a Boolean satisfiability problem (SAT) to be solved. The Alloy programming language is based on first-order logic and allows the specification and model checking of complex structures and behaviours. We discuss our verification process while walking through a few important features of Alloy, namely: *signatures*, *constraints* and *commands*. Our Alloy model file is provided as Listing B.1 in Appendix B on page 180.

Alloy signatures The Alloy programming language is based on sets and relations over these sets, which are declared via *signature* declarations. We describe a memory event as a signature (lines 33 to 43). In this signature, we declare four sets: memory events, memory event types, atomic events and consistency modes. We also declare five relations over these events: program order (*po*), control dependencies (*cd*), read-modify-write (*rmw*), same thread (*sthd*) and same location (*sloc*).

Alloy constraints Another feature of the Alloy programming language is *constraints*. There are several types of constraints: facts, functions, predicates and assertions. Facts are constraints that are assumed to always hold. Based on a memory event's sets and relations, we apply several facts about them. These facts define how different memory events relate to each other (lines 43 to 78) and also how the the different relations related to each other (lines 78 to 101). Functions and predicates are constraints that are not facts, *i.e.* they only hold when invoked and thus can be included (or excluded) for different configurations. When invoked, a predicate is a constraint that applies to a model. A function packages a predicate for re-use, *i.e.* it is an expression, which require input arguments and provides a result. For further details on functions and predicates, please refer to [121, §4.5].

Modelling the C memory model We combine signatures, predicates and functions extensively to verify our methods. We set up a top-level predicate `find_bug_chap3` (lines 273 to 288), where we describe two models. The first is the C11 memory model, based on the rules of Lahav *et al.* (line 282), and the second is a model of our method, which includes

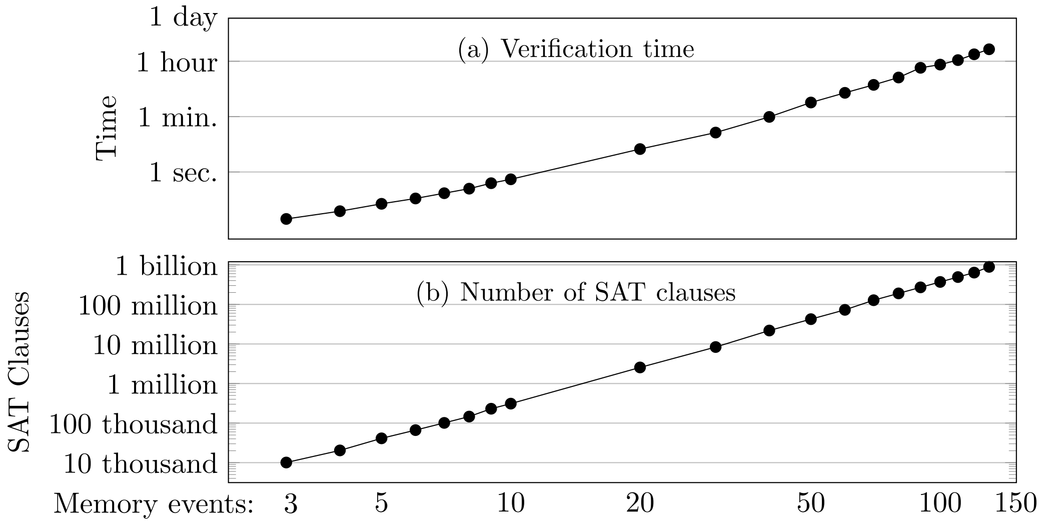


Figure 3.12: Verifying *mem-weak-fence* from (3.6) of page 73 up to a bounded number of events

the Shasha-Snir rules and our scheduling rules (line 285). Finally, we verify our scheduling rules by setting up these two models to generate counter-examples that are allowed by our method but disallowed by C11.

Modelling our scheduling rules Our scheduling rules, represented by *mem-weak-fence* of (3.6) from page 73, are directly translatable to Alloy constraints, as described by the function `rules_chap3` on line 244. Constraints are applied to atomics and fences individually. Lines 246 to 252 represent *mem-weak* of (3.5) from page 71. Line 246 represents *mem-alias* and *rar*, line 249 represents *acq*‡ and *sc*‡, and line 252 represents *rel*† and *sc*†. Finally, lines 256 to 264, represent *sc-fence*, *acq-fence* and *rel-fence* respectively.

Executing our verification via Alloy commands Finally, we execute a *command* over our top-level predicate `find_bugs_chap3`. A command executes a predicate to find instances that satisfy it. Lines 289 to 296 show examples of *run* commands, where we indicate the predicate to execute and its scope. The *scope* bounds the instance size, since Alloy requires a finite bound on its search space.

Verification results We run Alloy on a machine with 16 cores of 2.1 GHz AMD Opteron processors and 128 GB of RAM, and we use the Glucose SAT-solving backend. Alloy was able to verify that no counter-examples can be generated for executions of up to 130 memory events. Though our verification result is bounded, it is a strong result because many common memory-related bugs can be minimised to even smaller programs, typically comprising between four and six events [126]. Fig. 3.12(a) shows that Alloy's verification time increases cubically with the number of memory events. Alloy executes its verification in two steps: first, it constructs a SAT problem and then it solves this problem. In our

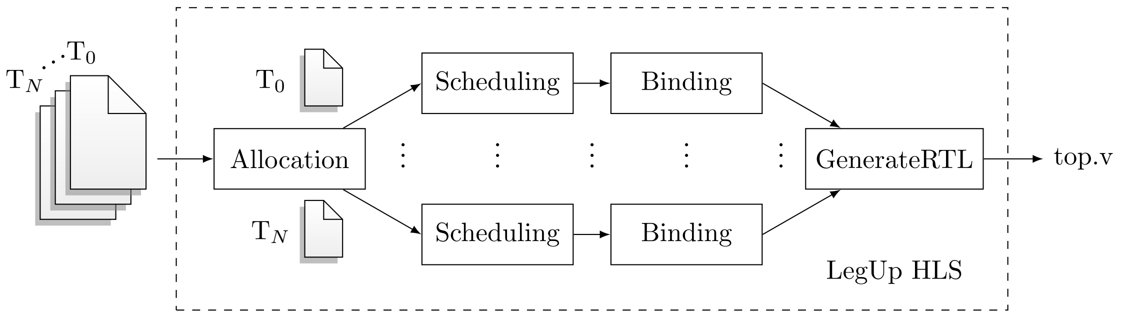


Figure 3.13: Stages of the LegUp Pthread's pure-hardware flow from its multi-threaded input (on the left) to a single RTL design (on the right).

case, the SAT construction dominates execution time since the number of SAT clauses generated by Alloy is also cubic with memory events, as shown in Fig. 3.12(b). Also, at 140 memory events, Alloy is unable to construct the SAT problem since it is too large to fit in memory.

3.7 Implementing our method on LegUp 5.1

We implement our method in LegUp 5.1 pthreads flow [8] (discussed thoroughly in §2.2.5 of page 39). LegUp's frontend automatically compiles a multi-thread C program with C11 atomics into LLVM IR. At the LLVM IR level, we have visibility of all memory operations within a thread and we identify the atomic operations and their consistency modes. Fig. 3.13 shows the LegUp tool flow. The Scheduling stage is where the scheduling rules generate SDC constraints (§2.2.6.1 of page 40). We augment this stage to inject additional memory dependencies of all atomics, on per-thread basis, based on our scheduling rules. We emphasise that despite the fact that we are injecting *intra-thread* constraints, our method ensures correct *inter-thread* synchronisation is preserved in the presence of C11 atomics. LegUp also serialises the execution of LLVM basic blocks, so we only need to inject our memory constraints within a basic block. Additionally, we also deal with fences at the RTL generation stage by simply removing them, since they are sufficiently handled by our constraints in §3.5.4 of page 72.

3.8 Extending LegUp to support atomic compare-and-swap

To support a larger set of benchmarks, we also needed to extend LegUp to support atomic compare-and-swaps (CAS) for the use of all our analyses throughout this thesis. As described in §2.3.3.2, CASes are read-modify-write operations that allow more than two concurrent threads to synchronise with each other. A CAS consists of an atomic load followed by an atomic store to the same location, which can not be interrupted. A straightforward method of implementing CAS operations in HLS is to perform an ordinary load and store while holding a mutual-exclusion lock [8]. However, as previously discussed, using locks to implement atomics is inefficient, since it requires extra cycles and prohibits reorderings. Instead of using locks, we modify LegUp to support CAS directly on hardware in two parts. Firstly, we ensure that our analyses can handle the mapping of CASes as a pair of atomic load and store operations connected by *rmw* edges. Secondly, we modify LegUp's RTL generator to implement the effect of a *rmw* edge, which ensures that no other aliasing memory operations can interrupt a CAS operation.

Analysis treatment of CASes C11 treats a CAS as a pair of accesses: an atomic load followed by an atomic store, as discussed in §2.3.3.2. Hence, we treat CASes as a pair of accesses where the consistency modes of the load and the store are determined from the consistency mode of the original CAS. 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 [101, §7.17.7.4.2]. We map the consistency modes of a CAS operation into different load and store event 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
- an *SC* CAS becomes an *SC* load and an *SC* store.

Any ordering edges that are associated to the either the load or store operations are automatically applied to both via the *rmw* edge.

Hardware implementation of CASes Figure 3.14 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. An uninterrupted sequence of read and write accesses to a shared memory location has the desired effect of a *rmw* edge. To achieve this effect, we first add circuitry that holds (`hold`) the arbiter's grant signal for an extra cycle, as shown in the grey shaded region of Fig. 3.14. 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

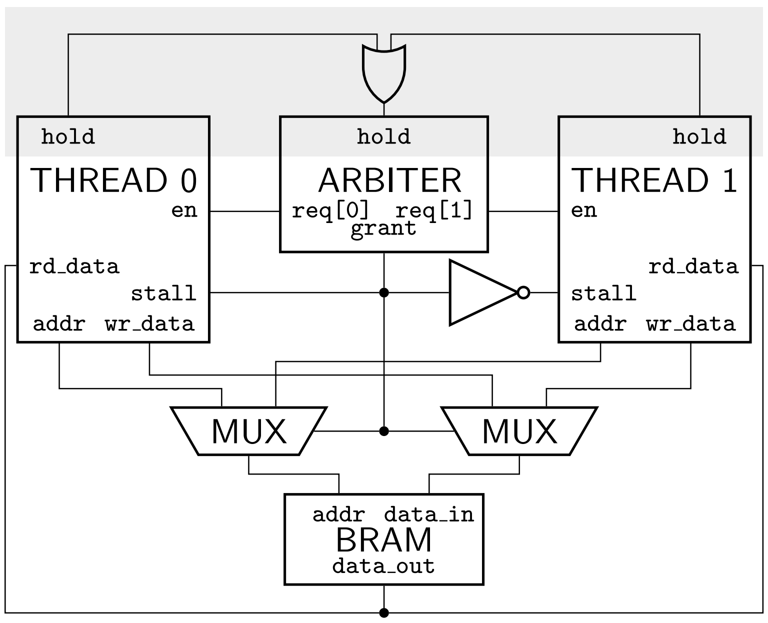


Figure 3.14: A (simplified) circuit showing how CAS works for two threads, the shaded region indicating circuitry added by us.

CAS can be packed into a single cycle whilst still preserving the functionality of a *rmw* edge.

3.9 Evaluation

In this section, we evaluate our method against the state-of-the-art support of atomics in LegUp 5.1. We evaluate our memory models on our set of memory-dominant experiments, discussed in §2.5.3.1. The rest of this section is as follows:

- In §3.9.1, we introduce the various design points for evaluation.
- In §3.9.2, we discuss the runtime performance of these design points for our set of memory-dominant experiments.
- In §3.9.3, we discuss the hardware utilisation overheads of these design points.

3.9.1 Design points

We evaluate six design points: one unsound version, two lock-based versions and three lock-free versions, as listed in Table 3.1. The *Unsound* design point is the performance of native LegUp HLS tool without any modification. Hence, this design point only preserves aliasing memory orderings. Although its results are invalid in a multi-threaded context, it serves as a practical upper bound for our design points.

The two lock-based versions, *OMP atomics* and *Locks*, use locks (via pthread mutexes) to achieve program correctness, since mutual exclusion is the only memory synchronisation

Table 3.1: Design points for evaluation of Chapter 3.

Short name	Description	Model	Ref.
Unsound	constraints treat atomics as ordinary accesses (theoretical upper bound)	*mem*	§3.2
OMP atomics	like *Unsound*, but each individual atomic access is wrapped around a lock	*mem*	
Locks	like *Unsound*, but each push/pop routine is wrapped around a lock	*mem*	
SC	constraints force all memory accesses to obey SC	*sc*	§3.5.1
SC atomics	constraints treat all atomics as if they are SC atomics	*mem-sc*	§3.5.2
Weak atomics	constraints sensitive to consistency modes of each atomic	*mem-weak*	§3.5.3

supported by LegUp. *OMP atomics* implements atomics by wrapping locks around each atomic access, which is similar to how OpenMP atomics are implemented in LegUp [68]. *Locks* implements lock-based programming by wrapping locks around each data structure routine, which is a coarser-grained approach of using locks than *OMP atomics*.

The three lock-free design points are based on our method of this chapter. The *SC* design point implements an SC memory model, which enforces memory serialisation. The *SC atomics* design point implements a memory model that supports SC atomics, where it treats all atomics as SC atomics. The *Weak atomics* design point implements a memory model that support weak atomics, where it is sensitive to their consistency modes.

Highlighting an interesting LegUp issue As discussed in §2.2.5.3, LegUp 5.1 implements a mutex's locking and unlocking functions as a spinning load and a store at IR level. This implementation causes an interesting problem, in particular the store. Since this store is a regular store, it can be reordered with other memory accesses. Hence, a mutex can potentially be released before the completion of a critical section.

There are two options to avoid this problem. The first option is to enforce SC, as we do with *SC*, to ensure that the mutex's store is not reordered. However, this approach enforces memory serialisation, which does not provide a good performance estimate of our lock-based design points. The second option is to disable if-conversion, which ensures that the mutex's store is its own basic block in LegUp and cannot be reordered with other memory accesses. Ultimately, the bullet-proof method of solving this problem is to write an LLVM pass to identify all mutex's store and enforce strict order on these operations. However, we choose the second option because it is easier to implement, ensures correctness and provides a fair performance estimate of our lock-based design points.

3.9.2 Performance

Fig. 3.15 shows the performance of our six design points on our set of memory-dominant experiments, which consists of nine unique settings, as discussed in §2.5.3.1. Our subfigures enumerate these settings, derived from combining three benchmarks (row-wise) and three data-flow patterns (column-wise). Note that we have 8 data points ($n = 8$) per design point. All averages reported are *geometric means*.

3.9.2.1 Unsound

Unsound has the best runtime performance across all experiments, since it only ensure ordering between aliasing memory accesses. Although its results are invalid, *Unsound* serves as an upper bound and our goal is to follow the trends of *Unsound* as closely as possible without violating correctness.

3.9.2.2 OMP atomics

OMP atomics is the state-of-the-art HLS method of implementing atomics but it is performs the worst across all experiments. On average, *OMP atomics* is 30× slower than *Unsound*. Also, *OMP atomics*'s performance worsens as we scale the thread count, with a maximum of 162× slower than *Unsound*. *OMP atomics* performs badly for two reasons. Firstly, locks enforce the serialisation of all atomic memory accesses. Secondly, the locking and unlocking functions require additional cycles and hardware, which impacts cycle count and clock frequency. Hence, this design point is unnecessarily low performance.

3.9.2.3 Locks

Instead of wrapping locking and unlocking functions around each atomic access, *Locks* wraps these functions around each data structure routine. On average, *Locks* performs 2× better than *OMP atomics*, with a maximum of 4×. *Locks* reduce the granularity of mutual exclusion to serialise routines, instead of serialising all memory accesses. Hence, although parallelism across routines is disallowed, parallelism within routines is still allowed. Furthermore, coarser-grained mutual exclusion reduces the number of function calls, which improves cycle counts and clock frequency. In rare cases, *Locks* can be slower than *OMP atomics* but this typically happens at low thread counts where the impacts of memory and routine serialisation are indistinguishable, as in Figures 3.15(d) and 3.15(g). In summary, *Locks* is the best-performing lock-based design point.

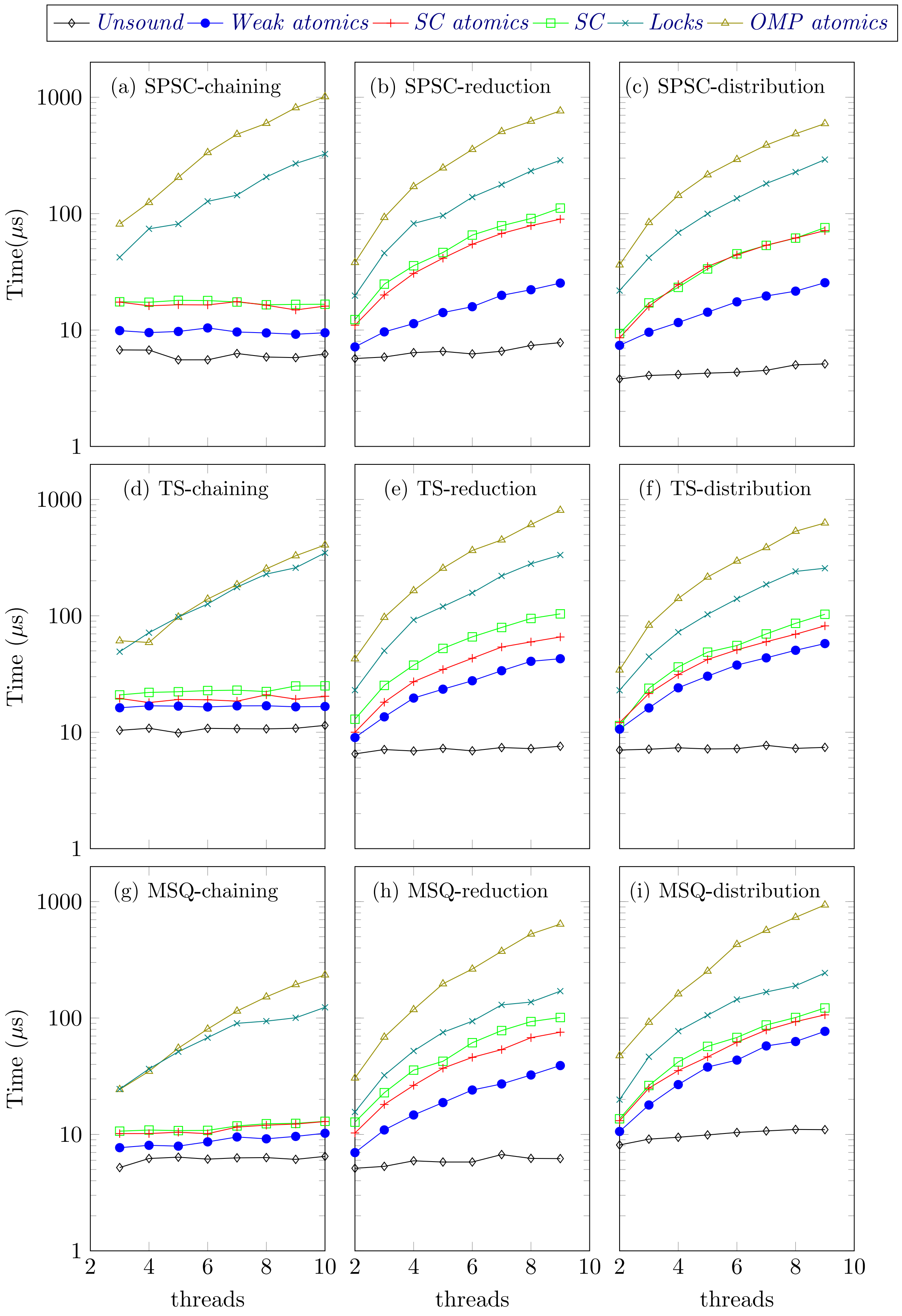


Figure 3.15: Runtime performance of our generated hardware for all design points Table 3.1 on our memory-dominant experiments.

3.9.2.4 SC

SC enforces memory serialisation within each thread of a concurrent program to ensure SC behaviour. Despite being the most conservative lock-free design point, *SC* performs 3× faster than *Locks*, on average, with a maximum of 20×. *SC* also scale better than *Locks* and *OMP atomics*, especially for the chaining experiments, since *SC* does not enforce memory serialisation globally but only locally within each thread.

3.9.2.5 SC atomics

The *SC atomics* implements all atomic accesses as SC atomics. On average, *SC atomics* is 1.2× faster than *SC*, with a maximum of 1.6×. In most cases, *SC atomics* runtimes are very similar to *SC*. This is due to the ratio of atomic accesses to non-atomic accesses in our benchmarks, as shown in Table 2.2. There are very few non-atomic accesses per data structure routine, leading to memory serialisation within each thread, just like *SC*. *SC atomics* also depends on the data-flow pattern, where *SC atomics* consistently outperform *SC* for all reduction experiments. Sometimes *SC* can outperform *SC atomics*, not due to number of cycles but due to clock frequency variations during hardware synthesis.

3.9.2.6 Weak atomics

Weak atomics implements atomics based on their consistency modes. As we can see in Table 2.2, none of the benchmarks use SC atomics by default. Hence, on average, *Weak atomics* is 1.6× faster than *SC atomics*, with a maximum of 3.8×. The different data-flow patterns also provide further insights.

The chaining experiments provide the smallest speedup, compared to *SC atomics*. In the chaining experiments, the number of routines within a thread is fixed (regardless of thread count). Hence, these experiments show the parallelism with routines of each benchmark. On average, *Weak atomics* improves the performances of the buffer, stack and queue by 2.2×, 1.3× and 1.4× respectively.

The reduction and distribution provide larger speedups, compared to the chaining experiments. For both of these experiments, the number of independent routines within the distributor or reducer threads increase with the thread count. Hence, *Weak atomics* is able to overlap these parallel routines to a certain extent. Although *Weak atomics* still scales with the thread count, its scaling is far better than *SC atomics*. Still, the best scaling achieved is by *Unsound* since it only preserve ordering of aliasing accesses.

Overall, *Weak atomics* is the best-performing design point. On average, *Weak atomics* is 6× faster than *Locks* (the best lock-based design). *Weak atomics* also support atomics correctly without violating correctness, unlike *Unsound*. Nevertheless, on average, *Weak*

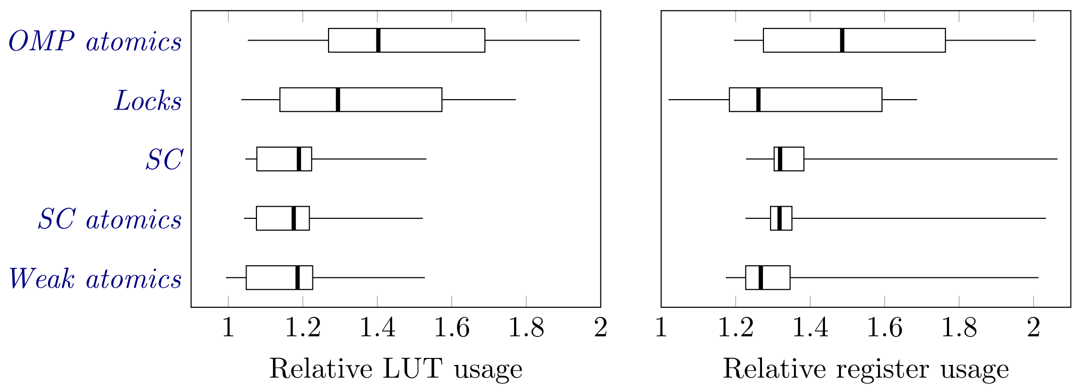


Figure 3.16: Relative resource utilisation of design points discussed in Table 3.1 compared to *Unsound*.

atomics recovers 40% of performance achievable by *Unsound*, with a maximum of 80%.

3.9.3 Resource utilisation

Fig. 3.16 shows the relative resource utilisation of our design points compared to *Unsound*. We divided each data point with the *Unsound*'s LUT and register usage, where each data point is distinguished by benchmark, data-flow patterns and thread count. We see that all design points require higher LUT and register usage compared to *Unsound*.

LUT usage *SC*, *SC atomics* and *Weak atomics* have similar LUT usage overhead, which is around 18% more than *Unsound*. LUT usage is proportional to schedule latency and typically these design points have higher latencies than *Unsound*. Both lock-based design points, *Locks* and *OMP atomics*, require higher LUT usage, compared to all lock-free design points. This trend is expected since additional hardware is required to implement the mutex hardware and its function calls within each thread. *OMP atomics* require more LUT usage than *Locks* since it requires more function calls. On average, *OMP atomics* requires an average of 9% more LUTs than *Locks*.

Register usage On average, *SC*, *SC atomics* and *Weak atomics* have similar register usage, which is around 30% more than *Unsound*. These three design points also have the higher maximum relative usage, which is about 2× more than *Unsound*, since the synthesis tools can identify the potential for register duplication to produce better clock frequencies. Interestingly, on average, *Locks* uses fewer registers than *SC*, *SC atomics* and *Weak atomics* since register duplication does not applied to it. *OMP atomics* uses the most registers since its schedule latencies is the highest.

3.10 Conclusion

In this chapter, we set out to address the question of devising a HLS-friendly method to implement fine-grained memory synchronisation of concurrent C programs. We motivated why the HLS memory models are insufficient to support C11 atomics. Then, we showed how we can enlarge its memory orderings to support atomic operations, by implementing additional *intra-thread* scheduling rules. By doing so, we can also support weak atomics and memory fences. Since implementation of weak atomics is complex and error-prone, we also ensured the correctness of our method by verifying our scheduling rules against C11 via automated model checking. Finally, we implemented our method on LegUp's scheduler and evaluated all our methods on a set of experiments. Overall, this chapter addresses research questions **RQ 1** and **RQ 2** that we presented in the Introduction chapter.

$$\begin{tikzcd}
\textit{OMP atomics} \arrow[r, "2\times"'] & \textit{Locks} \arrow[r, "3.1\times"'] & \textit{SC} \arrow[r, "1.2\times"'] & \textit{SC atomics} \arrow[r, "1.6\times"'] & \textit{Weak atomics} \arrow[r, dashed, "2.5\times"'] & \textit{Unsound}
\end{tikzcd}$$

Figure 3.17: Average speedups of our design points in Tab. 3.1 for our set of experiments.

Fig. 3.17 shows the summary of results for the six design points in Table 3.1. We see that the current LegUp mechanism of implementing atomics, *OMP atomics*, performs the worst since it implements atomics using locks. Instead, we implement *Locks* that performs 2×better than *OMP atomics*, since we serialise routines rather than serialise all memory accesses. We also show that serialisation of memory accesses within a thread without locks, *SC*, is 3.1×faster than *Locks*. This is our first encouraging results, which shows that posing atomics as a scheduling problem improves performance drastically. Furthermore, when we make our memory model sensitive to atomics via *SC atomics*, which treat all atomics as SC atomics, we achieve a further 1.2×speedup. Finally, we show that making our memory model sensitive to the consistency modes of each atomics via *Weak atomics* enables a further 1.6×speedup. Looking ahead, *Weak atomics* is still slower than our upper bound, *Unsound*, by 2.5×. Hence, there is still room for improvement and the following chapters (Chapter §4 and §5) aim to address this gap in different contexts.

4. Concurrency-aware Scheduling of Fine-grained C Concurrency

4.1 Introduction

In the previous chapter, we showed how *fine-grained* atomics can be synthesised efficiently via HLS scheduling constraints, instead of wrapping locks around them. Our method that support SC atomics is 7.5× faster than the HLS state-of-art on our set of experiments. Furthermore, since we have full control over the scheduling constraints of each atomic operation, we can also synthesise *weakly consistent* atomics directly onto hardware. We showed that weak atomics improves performance by 1.6×, compared to SC atomics, on our set of experiments.

In Chapter 3, we leverage the fact that it is standard practice for HLS tools to perform memory scheduling of concurrent C programs on a per-thread basis. Consequently, the memory orderings preserved within each thread is only based on its own memory access patterns *i.e.* thread-local analysis. Hence, we took advantage of this approach to enlarge the set of memory orderings within each thread. Despite only enforcing *intra-thread* memory constraints, we were able to ensure that the correct execution of atomics globally. C memory model allows global synchronisation to be achieved by simply enforcing memory orderings within each thread, as discussed in §2.3.3. In this chapter, we argue that thread-local analysis is a good first step but it leads to overly-conservative memory scheduling of threads, when we consider concurrent programs as a whole.

Consider the following single-threaded program, which consists of a 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. However, if we perform thread-local memory scheduling on this program then we have to work on the assumption that there might be other threads concurrently accessing `x` and `y`, and hence we 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)

In this case, the two stores must no longer be reordered. This is because the C memory model dictates that when an atomic load observes an atomic store in another thread, the two threads synchronise [101, §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. Hence, thread-local scheduling must account for this worst-case scenario and disallow reordering even for Program 1.

Natively, the C memory model defines memory access behaviour globally. C defines that atomic accesses act as explicit synchronisation points across threads and that any other memory accesses are restricted to synchronise as a consequence of these atomic synchronisation points. This definition implies that two memory accesses can synchronise during runtime when there exists a path of atomic synchronisation points between them. We utilise this definition to improve memory scheduling of individual threads, by only preserving orderings between pairs of memory accesses that are part of these synchronisation paths.

Hence, in this chapter, we propose a global analysis that determines which pairs of memory operations must not be reordered within a thread. We can perform such an analysis via HLS since HLS enforces that threads to be synthesised as hardware are identified via configuration settings/pragmas [3, 68, 120]. Hence, the entire program's memory access pattern is available at compile-time as inputs to our global analysis. Although our analysis receives a program's global access pattern as its input, it still generates *intra-thread* memory constraints as its output, which is then applied to memory scheduling of individual threads. Our analysis can handle programs that use both SC and weak atomics, and hence is verified via the Alloy model checker to build confidence of our method.

In this chapter, we discuss the following:

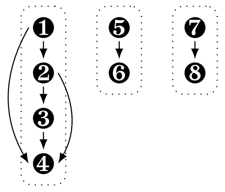
- In §4.2, we present a more practical example of why global analysis is beneficial when scheduling fine-grained C concurrency.
- In §4.3.1, we present our method that globally-analyses concurrent C programs with SC atomics and generates *intra-thread* scheduling constraints for each thread.
- In §4.3.2, we further extend our method to support *weak atomics*, by making our synchronisation paths sensitive to the consistency modes of atomics.
- In §4.4, we discuss an optimisation that reduces the number of synchronisation paths

`int xd=0,yd=0; atomic_int xr=0,yr=0;`

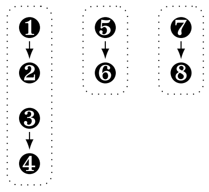
T0	T1	T2
❶ `xd=42;`	❺ `r1=atomic_load(&xr);`	❼ `r3=atomic_load(&yr);`
❷ `atomic_store(&xr,1);`	`if(r1==1)`	`if(r3==1)`
❸ `yd=17;`	❻ `r2=xd;`	❽ `r4=yd;`
❹ `atomic_store(&yr,1);`		

`assert((r1==1` $\implies$ `r2==42) && (r3==1` $\implies$ `r4==17))`

(a) a program



(b) thread-local analysis



(c) global analysis

Figure 4.1: Three-threaded message passing example with two channels.

to explore whilst obtaining the same set of constraints as our naïve method.

- In §4.5, we also check the correctness of our method and optimisation via automated model checking to ensure that they do not violate the C memory model.
- In §4.6, we discuss how we implement our method on LegUp 5.1 as an LLVM pass.
- In §4.7, we evaluate our global analysis approach against the thread-local analysis, from the previous chapter, on our memory-dominant experiments.
- Finally, in §4.7.4, we discuss the analysis times and scalability of our global analysis.

4.2 A motivating example

In this section, we provide a more realistic program that can benefit from our global analysis, and explain intuitively how our analysis works. Consider the program in Fig. 4.1(a), where `T0` uses atomic variables to pass messages to `T1` and `T2` respectively. 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 (`xd` or `yd`) 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 final-state assertion 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. 4.1(b) show the ordering constraints that the thread-local analysis of Chapter 3 would impose. These constraints are injected on the basis that ❷ and ❹ are atomic, and hence must not be reordered with other memory accesses, which is is derived via *mem-sc* of (3.4) from page 70. This memory model forces all four memory operations to be serialised during scheduling.

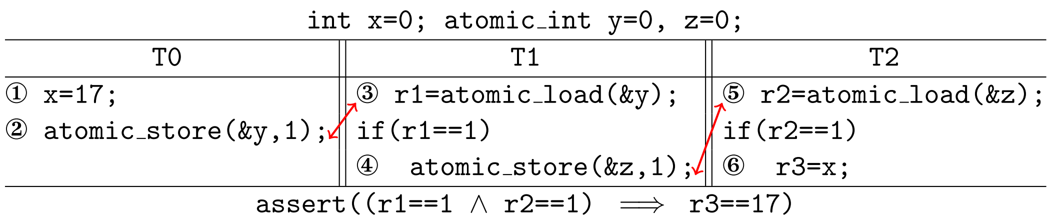
However, when considering the program as a whole, the ordering constraints in `T0` are overly conservative. Intuitively, there is no need to enforce ordering between the two independent message-passing channels within `T0` that transfer data to two different threads. Accesses to `xd` and `xr` transfer a message to `T1` whereas accesses to `yd` and `yr` transfer a message to `T2`. However, thread-local analysis of `T0` cannot make this judgement as it is agnostic to any behaviour outside its thread boundary. In contrast, our global analysis only considers pairs of atomics that can actually synchronise at runtime. As shown by the arrows in Fig. 4.1(a), `T0` can synchronise with `T1` and `T2` via their respective atomic synchronisation points *i.e.* if ❺ observes ❷ and if ❼ observes ❹ respectively. 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, where visibility only occurs between memory accesses to the same location. We can guarantee this visibility by preserving two ordering constraints in `T0` and one constraint each in `T1` and `T2`, as shown in Fig. 4.1(c). 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, affording better parallelism during memory scheduling.

4.3 Globally-analysing fine-grained C memory concurrency

In this section, we present our method for globally-analysing concurrent C programs with atomics to generate *intra-thread* memory constraints for memory scheduling. We can explore all possible *inter-thread* synchronisation opportunities by focusing on the atomic accesses of a concurrent program. When two atomic accesses synchronise at runtime, we must ensure that memory *visibility* is guaranteed *i.e.* all memory accesses that happened before this synchronisation point must be observable by all *aliasing* memory accesses that happens after this synchronisation point.

The inputs to our analysis are all memory operations of all threads that we grouped into several sets and relations, as discussed in §2.3.4 of page 52 (we shall provide a short recap when introducing them here). We treat each input memory operation as a memory event that can potentially occur at any point in time during hardware execution. Hence, we must ensure that we explore all possible permutations in which these memory events can execute at runtime during our global analysis. In the rest of section, we discuss the following:

- In §4.3.1, we present our global analysis that identifies explicit synchronisation points between atomics and presents a set of constraints when exploring all possible synchronisation paths of a given concurrent program;

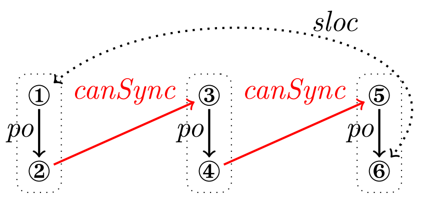


`int x=0; atomic_int y=0, z=0;`

T0	T1	T2
① `x=17;`	③ `r1=atomic_load(&y);`	⑤ `r2=atomic_load(&z);`
② `atomic_store(&y,1);`	`if(r1==1)`	`if(r2==1)`
	④ `atomic_store(&z,1);`	⑥ `r3=x;`

`assert((r1==1 ∧ r2==1) ⟹ r3==17)`

(a) program



(b) a legal path

Figure 4.2: An example of three-threaded message passing.

- In §4.3.2, we extend our analysis to be sensitive to weak atomics, which potentially reduces the number of synchronisation points and consequently improve memory scheduling of threads;
- Finally, in §4.3.3, we revisit our running example in the context of global analysis.

4.3.1 Identifying instructions that must not be reordered

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

$$\textit{canSync} = (V_{\text{at}} \times V_{\text{at}}) \cap \textit{sloc} \setminus \textit{sthd}. \tag{4.1}$$

The *canSync* relation connects any two atomic operations on the same location (*sloc*) from different threads (*sthd*). This definition treats all atomic accesses as SC atomics and hence utilises V_{at}, which the set of all atomics. Since *canSync* considers all atomic accesses of a concurrent program, it represents the global synchronisation opportunities across threads. 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, where visibility means a memory access that follows B must be able to see all accesses before A that are to the same location. For instance, the *canSync* edges of our motivating example are given by the arrows in Fig. 4.1(a). If ❷ synchronises with ❺ at runtime, then operation ❶ must be visible to ❻ since both these accesses are to the same location (`xd`) and hence both *po* edges (❶,❷) and (❺,❻) must be preserved.

In general, we must not only consider isolated *canSync* edges, but *paths* of them, in order to handle programs like the one shown in Fig. 4.2(a). In this program, 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 synchronisation can depend on paths that are made up of several *canSync* edges.

Therefore, to enumerate all possible synchronisation opportunities, we must explore and construct all possible synchronisation paths of a program, where each path obeys the following properties. 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:

$$\forall i.\, 0 \leq i \leq n \implies (v_i, v'_i) \in \mathit{po} \tag{4.2}$$

$$\forall i.\, 0 \leq i < n \implies (v'_i, v_{i+1}) \in \mathit{canSync} \tag{4.3}$$

$$(v_0, v'_n) \in \mathit{sloc} \tag{4.4}$$

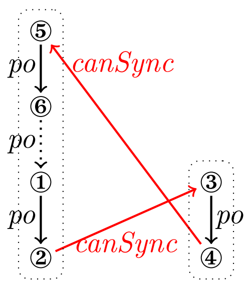
$$(v_0, v'_n) \in V_{\mathrm{ld}} \times V_{\mathrm{ld}} \implies (v_0, v'_n) \in V_{\mathrm{at}} \times V_{\mathrm{at}} \tag{4.5}$$

$$\forall i, j.\, 0 \leq i < j \leq n \implies (v_i, v_j) \notin \mathit{sthd} \tag{4.6}$$

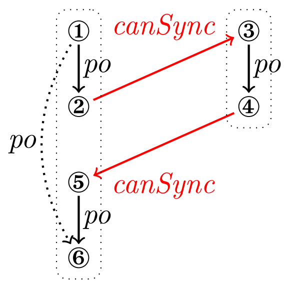
Condition (4.2) states that every path is an ordered list of $n + 1$ edges from *po*. Condition (4.3) states that the target operation of each *po* edge is connected to the source operation of the next *po* edge in the path via *canSync*. Also, we are only interested in paths that start and end with accesses to the same location (Condition 4.4) because visibility is only observed by memory accesses to the same location. Additionally, if a path begins and ends with loads, then we only consider it if both loads are atomic (Condition 4.5). Non-atomic loads can be reordered because this reordering cannot be observed unless the program has a data race [124], and this would be a programming error.

Fig. 4.2(b) shows a path that satisfies these four constraints. We have a path that begins and ends with *po* edges, with alternating *po* and *canSync* edges along the path, as stipulated by Conditions (4.2) and (4.3). Also, the start and end memory operations of the path are to the same location, *sloc*, and neither of them are non-atomic loads, as stipulated by Conditions (4.4) and (4.5).

Finally, we only consider paths that do not revisit a thread (Condition 4.6), since such paths are either illegal (cyclic) or can be minimised by removing the detour. Fig. 4.3 shows the two types of paths that we avoid with Condition (4.6). Fig. 4.3(a) shows that revisiting the same thread can result in a cycle, as we can construct paths from both ① to ⑥ and ⑥ to ①. This cycle forms an inconsistent execution, which violated program correctness, and hence it is illegal. Fig. 4.3(b) shows that revisiting the same thread can result in a redundant path of alternating *po* and *canSync* edges (solid lines), which can simply be minimised to one *po* edge from ① to ⑥ (dotted line). Condition (4.6) also



(a) Illegal path (`T2` inserted above `T0`)



(b) Redundant path (`T2` inserted below `T0`)

Figure 4.3: An example of paths that revisit the same thread.

induces a practical implication to our method, which is that it limits the path size, n, to be smaller than the number of threads. Hence, we can avoid any path explorations that are beyond the size of n.

We enumerate all paths that satisfy Conditions (4.2) to (4.6) and we preserve all the *po* edges that appear in at least one path. That is, we define the preserved program order, *ppo*, as follows:

$$ppo = \{(v, v', 0) \mid \exists p \in \mathit{AllPaths}.\, (v, v') \in p\}. \tag{4.7}$$

As an example, there are two paths in Fig. 4.1(a) that satisfy Conditions (4.2) to (4.6): [(❶, ❷), (❺, ❻)] and [(❸, ❹), (❼, ❽)]. Hence, the only edges that need to be preserved in Fig. 4.1(a) are the four *po* edges in its two paths: (❶,❷), (❺,❻), (❸,❹), and (❼,❽), as shown in Fig. 4.1(c).

4.3.2 Exploiting weak concurrency

Thus far, our analysis considers all atomics as SC atomics and hence are subjected to the same synchronisation opportunities. However, this is a conservative assumption since the C memory model limits their synchronisation opportunities of weak atomics, which in turn allows weak atomics to exploit better performance compared to SC atomics. Now, we describe how to extend our analysis to exploit weak atomics.

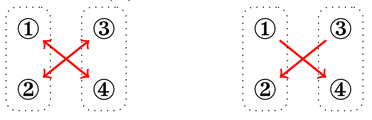
Recall that V_{sc}, V_{acq} and V_{rel} are the set of sequentially-consistent, acquire and release atomics, as in §2.3.4. Then, we redefine the pairs of atomics that can cause threads to synchronise, as follows:

$$\mathit{canSync} = ((V_{\mathrm{rel}} \times V_{\mathrm{acq}}) \cup (V_{\mathrm{sc}} \times V_{\mathrm{at}}) \cup (V_{\mathrm{at}} \times V_{\mathrm{sc}})) \cap \mathit{sloc} \setminus \mathit{sthd}. \tag{4.8}$$

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 atomics, or (b) either operation is an SC atomic. Condition (a) captures the one-way nature of release/acquire synchronisation [101, §5.1.2.4.11], while Condition (b) lets SC atomics retain their full

```
                        atomic_int x=0, y=0;
① atomic_store_explicit(&x,1,     ③ atomic_store_explicit(&y,1,
    memory_order_release);            memory_order_release);
② r1=atomic_load_explicit(&y,     ④ r2=atomic_load_explicit(&x,
    memory_order_acquire);            memory_order_acquire);
```

(a) a program



(b) SC (c) Weak

Figure 4.4: The 'store buffering' programming pattern, and its *canSync* edges under SC and weak consistency

Cycle:	1
`a=42;`	st$_{\texttt{na}}$ `a` 42
`ast(&b,1,REL);`	st$_{\texttt{REL}}$ `b` 1
`x=17;`	st$_{\texttt{na}}$ `x` 17
`ast(&y,1,REL);`	st$_{\texttt{REL}}$ `y` 1

Figure 4.5: A running example of four non-aliasing stores and its schedule based on global analysis when considering this thread as the entire program.

synchronising abilities.

Figure 4.4(a) shows an example program that is influenced by this logical strengthening of the *canSync* relation. It illustrates the 'store buffering' pattern, which appears in, for instance, Dekker's algorithm for mutual exclusion [127]. It consists of two atomic locations, `x` and `y`, with a release store and then an acquire load to these two locations in alternating order within each thread. If SC behaviour is enforced, then the outcome `r1` = `r2` = 0 would be forbidden by C. To ensure that this outcome cannot happen, our original analysis generates *canSync* edges, as shown in Fig. 4.4(b), which lead to paths such as [(①, ②), (③, ④)] thereby requiring to preserve both *po* edges. However, by re-defining *canSync* to be sensitive to weak atomics, it produces one-way *canSync* edges, as seen in Fig. 4.4(c), which in turn rules out any synchronisation paths. Hence, the two operations in each thread can be reordered since SC behaviour need not be preserved.

4.3.3 Revisiting our running example from §3.4

Now, we revisit our running example in the context of global analysis. This example was introduced in §3.4, where we have four non-aliasing stores of which two are release stores. Thread-local analysis has two possible outcomes: 1) latency of four cycles, if all atomics are SC atomics or 2) latency of three cycles, if we consider weak atomics.

Fig. 4.5 shows the schedule that global analysis generates if this is the only thread in the

Cycle:	1	2
❶ `a=42;`	$\mathrm{st}_{\mathtt{na}}$ `a` 42	
❷ `st(&b,1,REL);`		$\mathrm{st}_{\mathtt{REL}}$ `b` 1
❸ `x=17;`	$\mathrm{st}_{\mathtt{na}}$ `x` 17	
❹ `st(&y,1,REL);`		$\mathrm{st}_{\mathtt{REL}}$ `y` 1

Figure 4.6: A running example of four non-aliasing stores, of which two are store releases, and its schedule of based on global analysis when considering this thread as `T0` of Fig. 4.1(a).

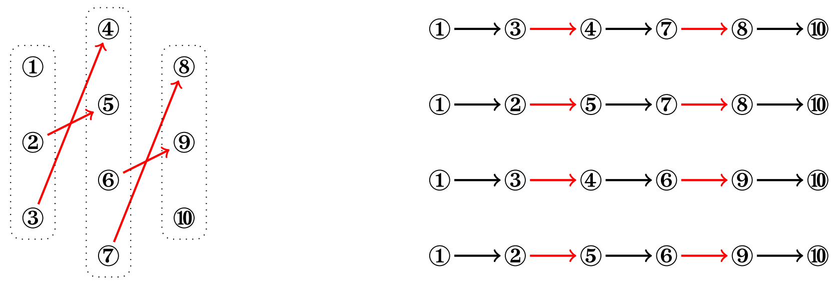
entire program. In a single-threaded program, there are no *canSync* edges and hence no paths to explore. Also, there are no aliasing accesses in this thread. Hence, all four stores can executed in parallel with a latency of one cycle. Although one can argue about the meaningfulness of such a single-threaded program, this toy example shows the advantage of global analysis, compared to thread-local analysis.

Fig. 4.1(a) represents a more meaningful program, where `T0` is writing to `T1` and `T2` via two independent message-passing channels. The memory accesses of `T0` are similar to the memory accesses of our running example, where `a` and `b` is one channel and `x` and `y` is another channel (as shown by the numberings to the left of the source code in Fig. 4.6). Our global analysis determines that only two *po* edges preserved within `T0`, which generates a schedule of two cycles. For this particular example, the consistency modes of atomics do not make a difference to the schedule latency. However, in our evaluation, we will see cases in which the use of weak atomics does make a difference in our evaluation. In summary, global analysis improves memory scheduling of this example by at least 1.5×, compared to weak atomics, and up to 2×, compared to SC atomics.

4.4 An optimised implementation of path enumeration

In §4.3.1, we described our method of enumerating the set of *AllPaths* and then extracting the *ppo* edges from it. Although our global analysis potentially improve memory scheduling, during initial implementations, we found that our method scales poorly with realistic programs. The crux of the problem is that realistic programs can have a large number of *canSync* edges and that can exponentially increases the number of exploration paths.

Hence, we devised an optimisation to improve scalability. We observed that in realistic programs many *canSync* edges overlap each other, which causes exploration of redundant paths. Hence, our optimisation avoids considering any overlapping *canSync* edges during path enumeration. However, removing these overlapping *canSync* edges does not capture all pairs of memory operations whose orderings must be preserved. Therefore, we propose a post-enumeration step that re-generates that same *ppo* as our naïve implementation. In



(a) Extracted information

(b) Results of path enumeration

Figure 4.7: An example of our naïve path enumeration.

summary, this optimisation is targeted to reduce the number of redundant paths visited during path enumeration, whilst ensuring the same *ppo* is generated.

More concretely, we 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*. The argument here is that these secondary *canSync* edges overlap with at least one other *canSync* edge and hence can be removed during the path enumeration and re-introduced after path enumeration to implicitly re-generates *AllPaths*, extracting *ppo* again.

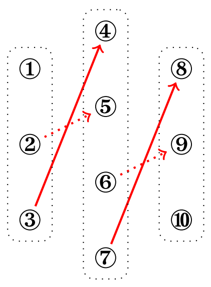
For example, Fig. 4.7(a) shows the shape of a program with three threads and four *canSync* edges. When we analyse this program using our naïve implementation, we see that there are four paths from ① to ⑩ as shown in Fig. 4.7(b). Out of these four paths, we only extract eight *ppo* edges, since some edges repeat themselves in different paths: (①,②),(①,③), (④,⑥),(④,⑦), (⑤,⑥),(⑤,⑦), (⑧,⑨) and (⑧,⑩). Although this example is small, we can see how the number of paths grow exponentially with the number of *canSync* edges and thread count. Our goal is to reduce the number of paths to enumerate and we can do so by analysing *canSync* carefully.

Consider the example in Fig. 4.7(a), but this time we identify the two dotted *canSync* edges as secondary edges, as shown in Fig. 4.8(a). We refer to these two edges as secondary 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. Hence, we only focus on enumerating paths based on primary *canSync* edges.

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

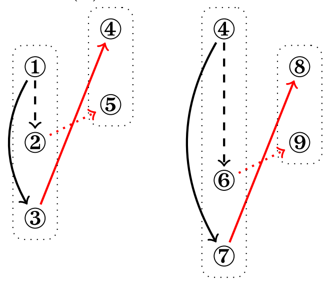
$$\begin{aligned}
&\mathit{canSyncPrimary} = \\
&\qquad \{(v_\mathrm{a}, v_\mathrm{b}) \in \mathit{canSync} \mid \nexists (v_\mathrm{c}, v_\mathrm{d}) \in \mathit{canSync}. \\
&\qquad\quad (v_\mathrm{a}, v_\mathrm{c}) \in \mathit{po}^* \wedge (v_\mathrm{d}, v_\mathrm{b}) \in \mathit{po}^* \wedge \\
&\qquad\quad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

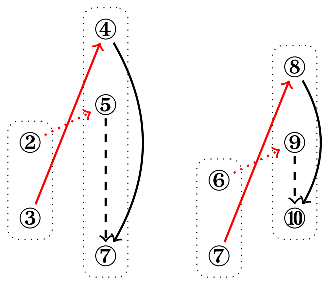
That is, $(v_\mathrm{a}, v_\mathrm{b})$ is a primary edge providing there exists no other *canSync* edge $(v_\mathrm{c}, v_\mathrm{d})$


(a) Extracted information

$$\begin{tikzcd}
① \arrow[r] & ③ \arrow[r] & ④ \arrow[r] & ⑦ \arrow[r] & ⑧ \arrow[r] & ⑩
\end{tikzcd}$$

(b) Results of primary path enumeration


(c) Secondary *canSync* exits above primary *canSync*


(d) Secondary *canSync* arrives below primary *canSync*

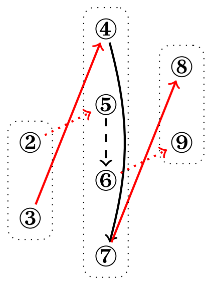

(e) Combination of both cases, (c) and (d)

Figure 4.8: An example of how our optimisation improves example in Fig. 4.7 where we consider primary (solid red) and secondary (dotted red) *canSync* edges.

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. (Recall 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 (4.3) to:

$$\forall i.\, 0 \leq i < n \implies (v'_i, v_{i+1}) \in \mathit{canSyncPrimary}. \tag{4.9}$$

For instance, our new definition only generates a single path from ① to ⑩, as shown in Fig. 4.8(b), compared to four paths when using our naïve implementation, as shown in Fig. 4.7(b). For now, we have achieved our goal of reducing the number of paths to explore but we have only extracted three out of eight *ppo* edges.

Having calculated the set of primary paths, it remains to generate the same *ppo* in a way that re-includes the non-primary paths. This can be done efficiently on an 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 secondary path between the same threads could have taken.

Intuitively, we must consider the effects of overlapping *canSync* edges on each enumerated path. In Fig. 4.8(b), we see that (②, ⑤) and (⑥, ⑨) overlap (③, ④) and (⑦, ⑧) respectively. These overlappings contribute to five *ppo* edges, which we represent as dashed black edges in Figures 4.8(c), 4.8(d) and 4.8(e). There are three overlapping cases:

- when a secondary *canSync* edge exits a thread above a primary *canSync* edge, as in

Fig. 4.8(c), where the *po* edge from the source of the primary *po* edge to the source of the secondary *canSync* edge must be preserved. Hence, (①, ②) and (④, ⑥) must be in *ppo*.

- when a secondary *canSync* edge arrives at a thread below a primary *canSync* edge, as in Fig. 4.8(d), where the *po* edge from the destination of the secondary *canSync* edge to the destination of the primary *po* edge must be preserved. Hence, (⑤, ⑦) and (⑨, ⑩) must be in *ppo*.
- and, finally, a combination of the first two cases, as in Fig. 4.8(e), where we might important corner cases if we do not consider the first two cases jointly. When we consider the first two cases together, we see that we must include (⑤, ⑥) in *ppo*.

All three cases only require processing a maximum of two primary *canSync* edges at a time, hence our post-processing step is local to each primary path and easy to implement.

More formally, we generate *ppo* as follows:

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

(recalling 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}).

In §4.7.4, we show the benefits of reducing the number of path explored during enumeration in terms of analysis times and comment on the complexity of this optimisation in §4.7.4.1.

4.5 Ensuring correctness via automated model checking

We re-use our Alloy models from the previous chapter, in §3.6, to verify our global analysis. In the previous chapter, we define two models: the C11 model and then Shasha-Snir rule, which we weaken to include our thread-local scheduling rules. Then, we set up Alloy to generate counter-examples in which a C11 execution is allowed by our rules but disallowed

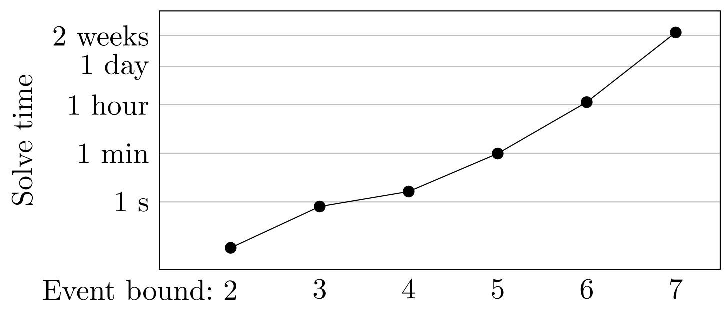


Figure 4.9: Time taken to verify our global analysis using Alloy.

by C11 and Alloy was not able to find any counter-examples for up to 130 memory events.

In this chapter, rather than modelling our thread-local scheduling rules, we replace them with our global scheduling rules that generate *ppo*. Hence, we weaken the Shasha-Snir rule in (3.7) of page 74 with constraints generated by global analysis as follows:

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

4.5.1 Modelling our global analysis on Alloy

To verify our global analysis, we first set up our top-level predicate in Alloy to find counter-examples that are allowed by our global analysis but forbidden by C11, expressed as `find_bugs_chap4` in lines 396 to 413 of Listing B.1 of Appendix B. Next, we must define *ppo* in Alloy and we do so in three parts.

Firstly, we define *canSync* which relates among events across threads in line 315, as defined in (4.8) of page 92. Secondly, we provide our path properties of Conditions (4.2) to (4.6) from page 91:

- Lines 340 to 353 represent Conditions (4.2) and (4.3), where paths must consists of alternating *po* and *canSync* edges that start and end with *po* edges;
- Lines 358 to 370 represent Conditions (4.4) and (4.5), where the first and last operations of a path must be to the same location, and if both of them are loads then they must be atomic;
- and, line 356 represents Condition (4.6), where no paths revisit the same thread.

Thirdly, and finally, we formalise our optimisation steps from §4.4. Line 334 defines *canSyncPrimary* of 95 from page 95, which reduce the number of paths to explore. However, when we use *canSyncPrimary* for path enumeration, we also need to post-process all paths to ensure that we generate the same *ppo*, as we should when using the entire *canSync* set, as described in lines 383 and 384.

Alloy was able to confirm that there are no buggy executions for any executions of up to seven events. Fig. 4.9 shows that the time taken for Alloy to deduce this result increases

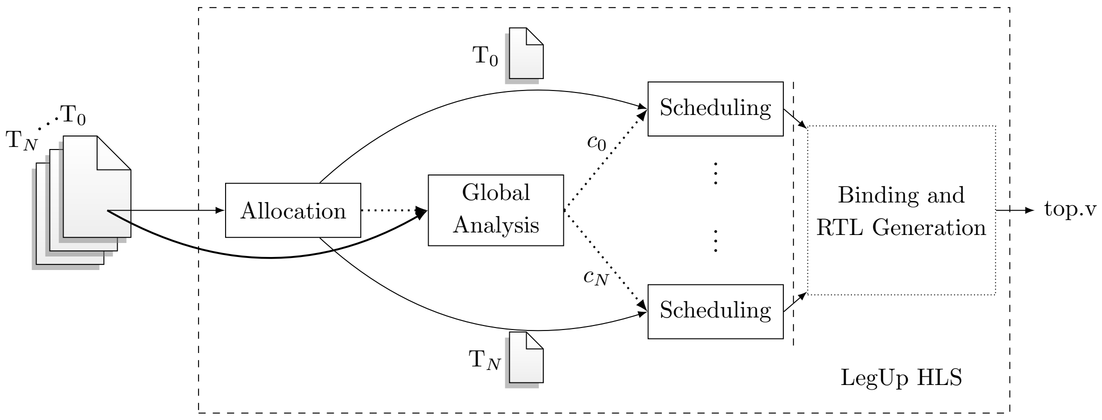


Figure 4.10: Our modification of LegUp pthread's pure-hardware flow to implement our method of global analysis for multi-threaded programs.

exponentially with the event bound. The exponential behaviour is expected as Alloy uses SAT solving, which is NP-hard. 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. Nevertheless, experience indicates that most bugs related to weak memory can be minimised to between four and six events [126], so our result is a useful and strong validation of our method. Also, the time taken by Alloy to verify seven bounds shows that our global rules are more complex than our thread-local rules. Given the same computational resources and time feasibility, Alloy can verify up to 130 memory events for thread-local analysis but only up to seven memory events for global analysis.

4.5.2 Comparing our global analysis to our thread-local analysis

In most cases, our global analysis imposes fewer constraints than thread-local analysis. However, there also exist programs for which our global analysis imposes *more* constraints than thread-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" [124]. 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 30 operations in about a second. The Alloy verification of these comparisons are included in Listing B.1 of Appendix B from lines 419 to 444.

4.6 Implementing our method in LegUp 5.1

We implement our global analysis in LegUp 5.1, where Fig. 4.10 shows our additions to the LegUp flow. To globally-analyse all threads, we cannot implement our analysis at the Scheduling stage as it only holds single-threaded information . Hence, we perform global analysis after the allocation stage as an LLVM module pass, which iterates over all LLVM functions and can access all global variables. Having obtained our inputs, we generate *canSync* and enumerate *AllPaths*, as discussed in §4.3.1, in a breath-first manner. At first, we generated paths of length one and then iteratively we generate paths of lengths up to the number of threads. We generate all paths that satisfy Conditions (4.3), (4.2) and (4.6) of page 91. Once we obtain these paths, we filtered out *ppo* by applying Conditions (4.4) and (4.5) of page 91, since these two conditions require the start and end of the paths to be identified. We also implement our optimisation, described in §4.4, within the same LLVM Module pass. Once we extract *Eppo*, we provide these constraints the Scheduling stage of their respective threads, as shown by the dotted arrows $c_0 \dots c_N$ in Fig. 4.10.

4.7 Evaluation

We evaluate global analysis, compared to thread-local analysis, on our memory-dominant experiments, as described in §2.5.3.1. In this section, we discuss the following:

- In §4.7.1, we present our design points for evaluation;
- In §4.7.2 and §4.7.3, we present the runtime performance and resource utilisation of our generated hardware;
- Finally, in §4.7.4, we discuss the scalability of our naïve and optimised implementations in terms of analysis times.

4.7.1 Design points

Tab. 4.1 summaries the five design points we evaluate, of which three are carried forward from Chapter 3: *Unsound*, *LocalSC* and *LocalWeak*. *Unsound* represents the results of our experiments when provided to LegUp without modifications, which is our theoretical upper bound. *LocalSC* is our thread-local analysis when all atomics are treated as SC atomics. *LocalWeak* is our thread-local analysis that supports weak atomics. Both *LocalSC* and *LocalWeak* as serve as baseline implementations of thread-local analysis. *LocalSC* and *LocalWeak* were referred to as *SC atomics* and *Weak atomics* in the previous chapter, in Table 3.1. We evaluate two design points based on global analysis: *GlobalSC* and *GlobalWeak*. *GlobalSC*'s *canSync* definition is based on (4.1) of page 90, where all atomics are treated as SC atomics. On the other hand, *GlobalWeak*'s *canSync* definition is based

Table 4.1: Design points for evaluation of Chapter 4.

Short name	Description	Model	Ref.
Unsound	constraints treat atomics as ordinary accesses (theoretical upper bound)	*mem*	§3.2
LocalSC	thread-local analysis that treats all atomics as if they are SC atomics	*mem-sc*	§3.5.2
LocalWeak	thread-local analysis that is sensitive to the consistency modes of atomics	*mem-weak*	§3.5.3
GlobalSC	global analysis that treats all atomics as SC atomics, where *canSync* is (4.1)	*ppo*	§4.3.1
GlobalWeak	global analysis that is sensitive to weak atomics, where *canSync* is (4.8)	*ppo*	§4.3.2

on (4.8) of page 92, which supports weak atomics.

4.7.2 Hardware Performance

Fig. 4.11 shows the runtime performance of our five design points on our memory-dominant experiments. All averages reported are geometric means.

4.7.2.1 Recap of *Unsound*, *LocalSC* and *LocalWeak*

We recap a few highlights of thread-local analysis. We see that *LocalSC* is the slowest across all experiments, since it implement all atomics as SC atomics. *LocalWeak* always performs faster than *LocalSC*, since weak atomics requires fewer ordering constraints and hence achieves better memory parallelism. Also, both *LocalSC* and *LocalWeak* do not scale well with the thread count for the reduction and distribution experiments. As the number of threads increases, the distributor (or reducer) thread in the distribution (or reduction) experiments starts to access multiple independent routines. However, since thread-local analysis is agnostic to the existence or behaviour of other threads, it limits the overlapping of these routines. In contrast, we see that the *Unsound* design point scales perfectly for all experiments, since *Unsound* only preserves memory ordering between aliasing operations.

4.7.2.2 *GlobalSC*

GlobalSC* versus *LocalSC *GlobalSC* is always faster than *LocalSC*. *GlobalSC* improves both parallelism within routines and across routines, since it is 1.3× faster (with a maximum of 1.6×) for the chaining experiments and it is 3.8× and 3.4× faster (with a maximum of 8× and 7.4×) for the reduction and distribution experiments. Parallelism across routines allows for better speedups, compared to parallelism within routines, because *GlobalSC* scales well with the thread count, just as *Unsound*.

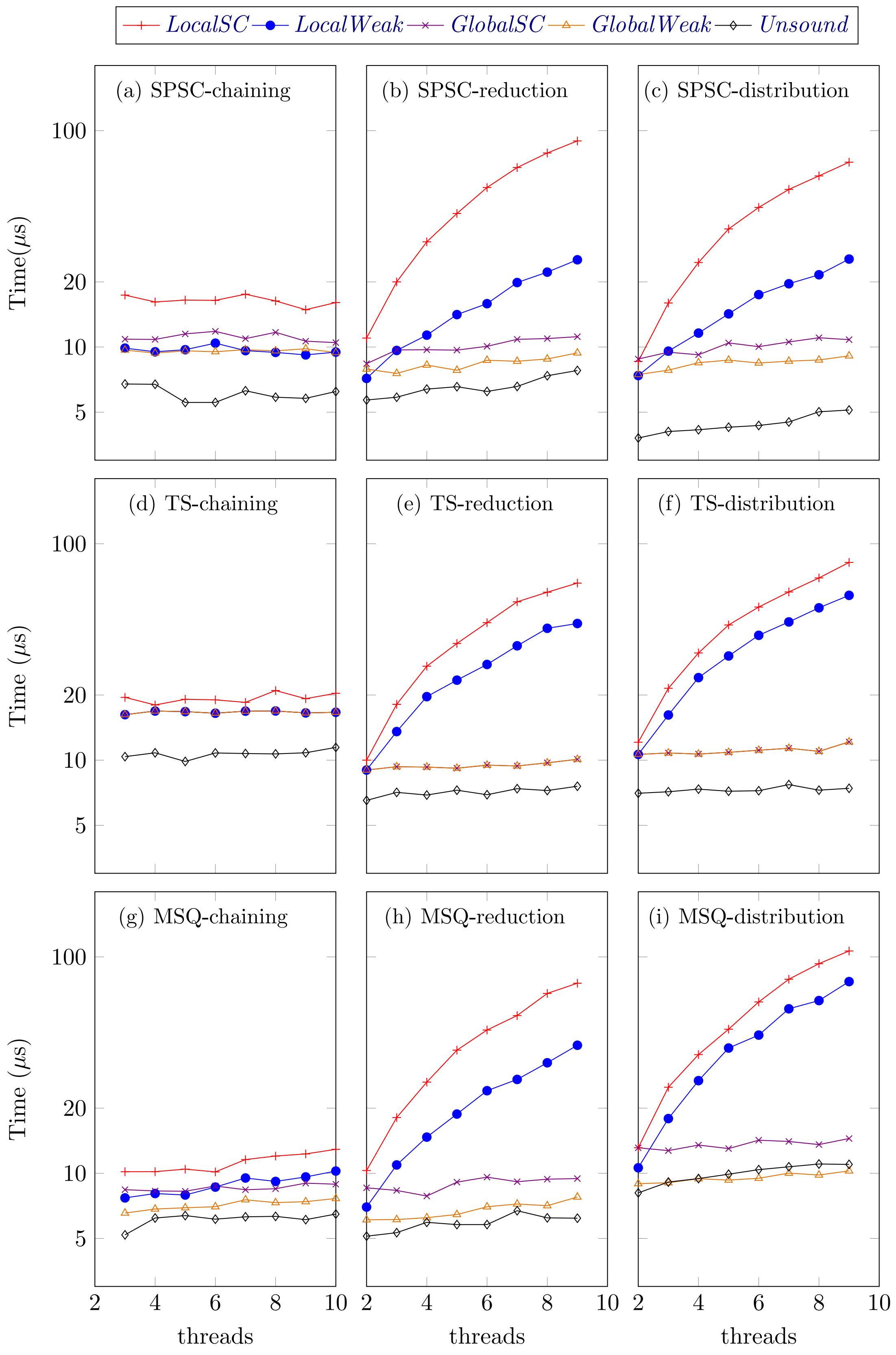


Figure 4.11: Runtime performance of our generated hardware for all design points in Table 4.1 on our memory-dominant experiments.

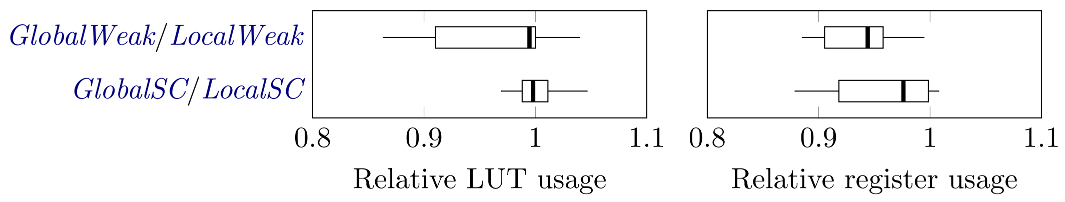


Figure 4.12: Relative resource utilisation of global analysis, compared to thread-local analysis, of both SC atomics and weak atomics.

4.7.2.3 *GlobalWeak*

GlobalWeak* versus *LocalWeak Global analysis also improves performance of programs that use weak atomics. In most cases, *GlobalWeak* is faster than *LocalWeak*. *GlobalWeak* is 1.1× faster than *LocalWeak* for chaining experiments. *GlobalWeak* is 2.3× and 2.6× faster than *LocalWeak* for reduction and distribution experiments, with a maximum of 5× and 7.5×. Once again, these experiments show that global analysis improves both the intra-routine and inter-routine parallelism.

GlobalWeak* versus *GlobalSC *GlobalWeak* further exploits further parallelism within routines afforded by weak atomics, compared to *GlobalSC*, and it is 1.2× faster than *GlobalSC*. These speedups also vary with benchmarks. For the buffer and queue experiments, *GlobalWeak* is faster than *GlobalSC* by 1.2× and 1.3×. In contrast, *GlobalWeak* and *GlobalSC* perform similarly for the stack experiments. This can be related to the ratio and consistency modes of the each benchmark's memory accesses, as characterised in Table 2.2. The buffer and queue have non-atomic accesses and also use weak accesses. In contrast, the stack only uses atomic accesses and its weak accesses does not permit further reorderings, compared to SC atomics.

GlobalWeak* versus *Unsound In most cases, *GlobalWeak* is slower than *Unsound*. The only exception is the queue's distribution experiments in Fig. 4.11(i), where *GlobalWeak* is slightly faster than *Unsound*. The schedule latencies of both *GlobalWeak* and *Unsound* are the same, so one would expect their cycle counts to be the same. Although their schedule latencies are the same, the cycle count can differ at runtime because the CASes can stall for a few cycles when there is contention. Hence, due to dynamic interactions of the CAS accesses, it is possible that *GlobalWeak*'s cycle count is marginally small than *Unsound*. Their cycle count difference is never larger than the number of iterations, which suggests that the CASes do not stall for more than one cycle per iteration.

4.7.3 Resource Utilisation

Fig. 4.12 shows the relative LUT and register usage generated by global analysis, compared to thread-local analysis. Overall, we see that the impact of global analysis on resource utilisation is minimal since it incurs similar amount of LUTs, compared to thread-local analysis. LUT usage relates to schedule latencies. Since *GlobalSC* does not reduce schedule latencies much, its LUT usage range is shorter than *GlobalWeak*. In contrast, we see that *GlobalWeak* can save up to 14% relative LUT usage, since is capable of achieving shorter latencies. On average, global analysis require 5% fewer registers, compared to thread-local analysis, and can save up to 12% registers.

4.7.4 Analysis scalability

Finally, we discuss the time taken by our naïve and optimised implementations of global analysis to generate memory constraints of all threads for all our experiments. Fig. 4.13 shows the runtimes of our implementations, which has the following four design points:

- *NaïveSC* implements *GlobalSC* naively by exploring *AllPaths*, as presented in §4.3.1.
- *NaïveWeak* is similar to *NaïveSC* but uses the *canSync* of (4.8) from page 92.
- *OptimisedSC* implements *GlobalSC* by exploring the optimised implementation that only enumerates *PrimaryPaths*, as discussed in §4.4, and then re-constructs *ppo*.
- *OptimisedWeak* is similar to *OptimisedSC* but uses the *canSync* of (4.8) from page 92.

***NaïveSC* performs the worst** In most cases, *NaïveSC* is the worst-performing implementation because it explores paths based on the original *canSync* presented in (4.1), which treats all atomics as SC atomics. *NaïveSC* does especially badly for the chaining experiments, where it shows the exponential nature of path-enumeration. The higher the thread count, the more paths to explored and hence the larger the analysis time and memory usage. For these chaining experiments, we exhaust memory quickly (up to 6GB of RAM) since we implement a breath-first enumeration. For this reason, *NaïveSC* can only generate four and three data points for the stack and queue experiments respectively (Fig. 4.13(d) and Fig. 4.13(g)). We could implement our path enumeration as a depth-first exploration, but this would simply shifts the complexity problem to the time dimension, rather than the space dimension. We see that *NaïveSC* scales badly for all stack and queue experiments, since these benchmarks require atomic compare-and-swaps.

OptimisedSC* improves *NaïveSC We see that *OptimisedSC* copes well with all the worst-case behaviour of *NaïveSC*, discussed in the previous paragraph. In most cases, *OptimisedSC* is faster than *NaïveSC* because it reduces the exponential growth of path-enumeration. On average, *OptimisedSC* is 12× faster than *NaïveSC*, with a maximum of 1800×. Although our optimisation also consists of a post-enumeration step, the time savings obtained from reducing the number of paths to visit during enumeration far outweighs the cost of this post-enumeration step.

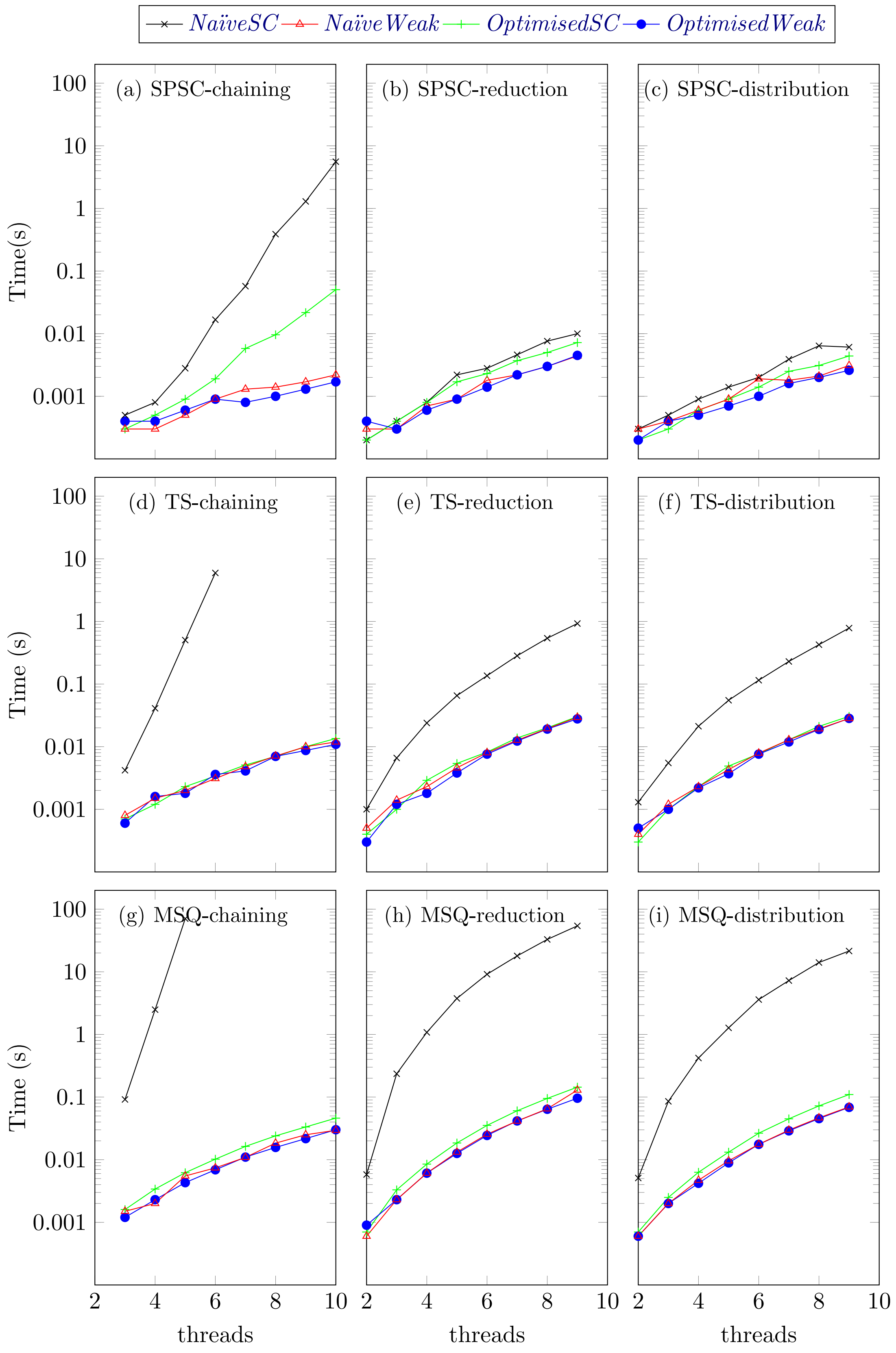
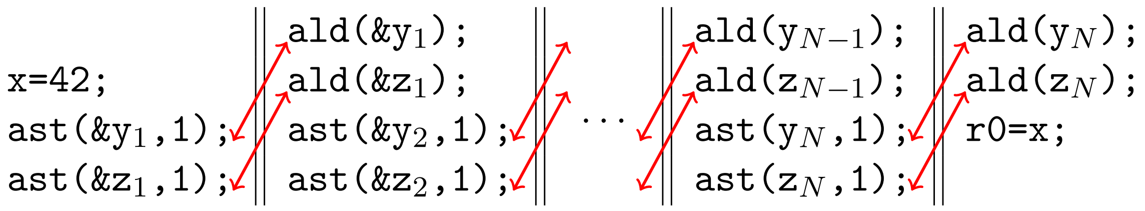


Figure 4.13: Analysis times of our naïve and optimised analysis.



```
                 ║ ald(&y1);       ║       ║ ald(yN−1);     ║ ald(yN);
x=42;            ║ ald(&z1);       ║       ║ ald(zN−1);     ║ ald(zN);
ast(&y1,1);      ║ ast(&y2,1);     ║  ...  ║ ast(yN,1);     ║ r0=x;
ast(&z1,1);      ║ ast(&z2,1);     ║       ║ ast(zN,1);     ║
```

Figure 4.14: A class of programs on which our analysis scales poorly, because the number of paths scales exponentially with the size of the program.

Weak atomics naturally reduce analysis times The runtime of *NaïveWeak* is much better than *NaïveSC*. This is because weak atomics, by definition, only highlights the important inter-thread synchronisations, reducing $|\mathit{canSync}|$. Hence, *NaïveWeak*'s path enumeration is less time-consuming compared to *NaïveSC*. On average, *NaïveWeak* is 16× faster than *NaïveSC*, with a maximum of 2500×. Occasionally, *NaïveWeak* can be slower than *NaïveSC*, but only for low thread counts and under the region of 10 milliseconds.

Effects of optimisation is less evident on weak atomics When we apply for our optimisation on programs with weak atomics, it benefits are less evident. Since weak atomics itself reduce the size of the *canSync* set, our optimisation does not impact runtime significantly as it does not eliminate many secondary edges from the naïve *canSync* set. Hence, on average, *OptimisedWeak* has similar runtime to *NaïveWeak*. *OptimisedWeak* can be up to 1.9× slower than *NaïveWeak*, since our optimisation imposes additional post-processing that is largely unnecessary for weak atomics.

4.7.4.1 Worst-case scalability

In the worst case, the runtime of our global analysis can still scale exponentially, since its complexity not only depends on the program size but also the access pattern of the input program. Fig. 4.14 shows a pathological program that scales exponentially, despite applying our optimisation. 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 synchronisation choices for each stage in the chain, either via a `y`-variable or via a `z`-variable.

4.8 Conclusion

In summary, we show that global analysis can improve memory scheduling of threads of a concurrent program. We do so by exploiting C11 definition that describes memory behaviour in terms of global executions, rather thread-local memory behaviour. Our global analysis can handle SC and weak atomics, as well as atomic compare-and-swaps. Since our

$$\begin{tikzcd}
\mathit{LocalSC} \arrow[r, "1.6\times"'] \arrow[d, "3.6\times"'] & \mathit{LocalWeak} \arrow[d, "1.9\times"] & \\
\mathit{GlobalSC} \arrow[r, "1.2\times"] & \mathit{GlobalWeak} \arrow[r, dashed, "1.4\times"] & \mathit{Unsound}
\end{tikzcd}$$

Figure 4.15: Speedups for design points in Table 4.1, averaged over all experiments.

analysis scope is global, our path enumeration can grow exponentially. Thus, we propose an optimisation to reduce the exponential growth of our analysis, especially for SC atomics. We also turn to automated model checking via Alloy to verify our method and ensure that our rules, and also our optimisation, is correct and that our generated hardware is correct by construction. We implement our method in LegUp 5.1 and compare our global analysis to our thread-local analysis from Chapter 3. Overall, this chapter primarily addresses research question **RQ 3**, and also **RQ 2**, that we presented in the Introduction chapter.

Fig. 4.15 provides a summary of how global analysis improves performance of our set of experiments, extended from §3.9. As discussed in the §3.10, thread-local analysis of weak atomics (*LocalWeak*) provides a 1.6× speedup, compared to thread-local analysis of SC atomics (*LocalSC*). Overall, global analysis achieves better runtimes than thread-local analysis on our set of experiments. A speedup of 3.6× is achieved when globally-analysing SC atomics (*GlobalSC*) compared to thread-local analysis of SC atomics (*LocalSC*). Also, a speedup of 1.9× is achieved when globally-analysing weak atomics (*GlobalWeak*) compared to thread-local analysis of weak atomics (*LocalWeak*). We see that *GlobalWeak* is also 1.2× faster than *GlobalSC*, since weak atomics can further exploit memory parallelism. Finally, we see that *Unsound* is only 1.4× faster than *GlobalWeak*. We have managed to bridge the gap between *LocalWeak* and *Unsound* which was 2.6×, as reported in Fig. 3.17. *GlobalWeak* is the closest we can get to *Unsound* without violating program correctness. In the next chapter, we look beyond this upper bound to further improve our runtimes.

5. Enabling Loop Pipelining for Fine-grained C Concurrency

5.1 Introduction

In Chapter 3 and 4, we presented methods to synthesise programs with fine-grained atomics via HLS scheduling. Thread-local analysis is where each thread's memory constraints is generated based only on its memory accesses, whereas global analysis is where each thread's memory constraints is generated based on the global access pattern. We show that thread-local analysis yields a speedup of 7.5×, compared to the state-of-the-art, for our set of experiments and that globally-analysing concurrent programs achieve a further speedup of 3.4×, compared to thread-local analysis. We also discuss that global analysis is close to our theoretical upper bound, which only preserves aliasing memory orderings. In this chapter, we present an effort to venture beyond this upper limit.

Thus far, both our thread-local and global analyses support code with loops but assume that operations of multiple loop iterations do not execute simultaneously. This assumption is conservative since HLS tools allows *loop pipelining*, where successive loop iterations can start before preceding iterations complete. To implement loop pipelining, a HLS tool computes an *initiation interval* (II), which is the interval between the start times of consecutive iterations. The tool then arranges operations of an iteration into a pipeline schedule that can be repeated at fixed II. The computed II and pipeline schedule is feasible if it does not violate any dependency and resource constraints. Smaller IIs increase instruction-level parallelism but also increases hardware utilisation and potentially decreases clock frequencies. Hence, loop pipelining presents an interesting trade-off between area and performance that relies on the generated II.

The goal of this chapter is to extend our thread-local and global analyses to support loop pipelining of concurrent programs with atomics. By supporting loop pipelining, we cannot only reordering memory operations within an iteration but across loop iterations, improving parallelism during memory scheduling. Whilst we encourage memory reorderings across iterations, we must ensure that the generated pipeline schedules do not violate program correctness. Currently, our scheduling rules from previous chapters are insufficient to support atomics in a loop pipelining context, since we assumed that loop iterations do not overlap and that all our constraints are *intra-iteration i.e.* its dependence distances are all zero. Therefore, in this chapter, we extend our analyses to also generate *inter-iteration* memory constraints for atomic accesses, thereby supporting atomics within pipelined memory schedules.

In terms of C11 execution, each memory operation of an iteration expresses itself as an independent memory event. However, C11 does not distinguish between memory events of different iterations. A memory event of an iteration is simply related to all memory events of consecutive iterations by program order (po). In contrast, HLS tools distinguishes memory constraints across iterations via dependence distance, which is the number of iterations between two memory operations. For our work, we only focus on preserving memory orderings of atomics between two consecutive iterations, since these orderings inductively extend to subsequent iterations. Hence, all our inter-iteration memory constraints have dependence distances of one. Several methods that analyse loop-carried dependences, especially via the polyhedral model [128, 129, 130], can generate memory constraints with dependence distance that is larger than one. It is important to note that our analyses does not restrict these methods, but simply complement the constraints generated by these methods.

Our work in this chapter is unique for three reasons. Firstly, since we synthesise atomics via HLS scheduling constraints, our method is the first method capable of supporting loop pipelining for atomics. In comparison, state-of-the-art HLS tools implement atomics via wrapping locks around them via function calls. Typically, these function calls must execute in-order, implicitly forcing loop iterations to also execute in-order and hence disabling loop pipelining. Secondly, our method is also the first method capable of synthesising weak atomics in the context of loop pipelining. Weak atomics affords better memory parallelism, compared to SC atomics, and this property also applies to memory operations across iterations. Finally, our method is generally applicable to any lock-free programs with atomics. Choi *et al.* [71] developed a method to pipeline lock-based concurrent programs. However, their method relies on users utilising their customised streaming library, which restricts the programming idioms that can generate high-performant streaming. Our method faces no such problem and can be applied to any C11 program with atomics.

In this chapter, we discuss the following:

- In §5.2, we formalise inter-iteration memory constraints generated by HLS tools.
- In §5.3, we demonstrate, by example, why these constraints are insufficient to support loop pipelining of atomics.
- In §5.4, we present our loop extensions for both our thread-local and global analyses.
- In §5.5, we re-visit our running example to contextualise how loop-pipelining benefits performance.

- In §5.6, we ensure that our extensions adhere to the C memory model.
- In §5.7, we discuss how we implement our methods in LegUp 5.1.
- Finally, in §5.8, we evaluate our extensions on two sets of experiments.

5.2 Formalising inter-iteration memory constraints of state-of-the-art

In §3.2, we formalise the memory constraints generated by current HLS tools in the absence of loop pipelining. *mem* of (3.2) from page 64 preserves the RAW, WAR and WAW dependencies of aliasing memory operations via *mem-alias* and disallows overlap of operations from different iterations via *nopipe*. However, all edges in *mem-alias* are intra-iteration, *i.e.* they all have a dependence distance of zero. To support loop pipelining, HLS tools eliminate *nopipe* and also enforce RAW, WAR and WAW dependencies of aliasing memory operations across iterations, as defined by *mem-pipe*:

$$\textit{mem-pipe} = \textit{intra-iter} \cup \textit{inter-iter} \tag{5.1}$$

where

$$\begin{aligned}\textit{intra-iter} &= \textit{mem-alias}\\ \textit{inter-iter} &= \{(v, v', 1) \mid (v, v') \in \textit{sloc} \wedge (v \in V_{\text{st}} \vee v' \in V_{\text{st}})\}.\end{aligned}$$

inter-iter expresses that for every memory operation v in the current iteration and every memory operation v' in the next iteration, where at least one of them is a store and both accesses are to the same location, there must exist an inter-iteration constraint of dependence distance one from v to v'. Together, *intra-iter* and *inter-iter* preserve RAW, WAR and WAW dependencies of aliasing memory operations within and across iterations. Note that it is possible that two memory operations can be represented in both *intra-iter* and *inter-iter* with different distances. In this case, the *intra-iter* edge takes precedence since it is stronger.

Similar to our discussion in §3.2, HLS tools enforce memory orderings between aliasing memory operations across iterations, permitting reorderings between non-aliasing memory operations and between operations that have RAR dependencies across iterations. On one hand, omitting these reorderings can improve parallelism during memory scheduling. On the other hand, omitting these reorderings can lead to incorrect behaviour when synthesising atomics in the context of loop pipelining. In the next section, we show an example of why *mem-pipe* cannot support loop pipelining for programs with atomics as it is stands, which motivates our work of enlarging it systematically.

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

```
    T0() {
     for(int i=1;i<=2;i++){
1.1   atomic_store(y, i, REL); ❶
1.2   int data = i;
1.3   atomic_store(x, data, RLX); ❷
     }
    }
```

```
    T1() {
2.1  int r0=atomic_load(y, ACQ); ❸
2.2  if(r0==2){
2.3   r1 = atomic_load(x, RLX); ❹
2.4  }
    }
```

assert($r0 == 2 \implies r1 \neq 0$)

Figure 5.1: A minimal example of message-passing across loop iterations.

Cycle:	1	2
1.1	st y 1 REL	st y 2 REL
1.3	st x 1 RLX	st x 2 RLX

Figure 5.2: Pipeline schedule generated by LegUp of `T0` in Fig. 5.1, where different shades represent different iterations and each store takes one cycle.

5.3 A motivating example

In this section, we show how current HLS tools synthesise atomics in the context of loop pipelining and why it violates program correctness. Fig. 5.1 is a two-threaded program with two atomic locations, `x` and `y`. `T0` comprises a two-iteration for-loop, where each iteration consists of a write to `y` and then to a write to `x`. The first write in the loop body is a release store of the loop index `i` to `y`. The second write is relaxed store of the loop index `i` to `x`. `T1`, which consists of straight-line code, executes an acquire load of `y` and if the value of this load of `y` is two, then it executes a relaxed load of `x` to check that its value is not zero (as enforced by the final-state assertion). This assertion must hold because, in `T0`, the relaxed store of `x` in the first iteration must happen before the release store of `y` of the second iteration. Therefore, if `r0` is two, then the release store to `y` of the second iteration has been executed and hence `r1` cannot be zero.

Fig. 5.2 shows the pipeline schedule that LegUp generates natively based on the code in `T0`. The darker shade represents operations from the first iteration and the lighter shade represents operations from the second iteration. LegUp preserves the order of aliasing memory operations within (*intra-iter*) and between (*inter-iter*) iterations, as described in *mem-pipe* of (5.1) from page 110. Within an iteration, the stores of `y` and `x` can be executed in parallel since they do not alias. Between iterations, the ordering of writes to aliasing memory operations across iterations are preserved. Hence, the writes to `y` and `x`, respectively, happen in iteration order. This pipeline schedule does not violate the program assertion regardless of when the observing thread `T1` chooses to execute its

```
                     atomic_int x = 0; atomic_int y = 0;
   T0(int arg) {                            T1() {
    for(int i=1;i<=2;i++){             2.1  int r0=atomic_load(y, ACQ); ❸
1.1  atomic_store(y, i, REL); ❶        2.2  if(r0==2){
1.2  int data = i/arg;                 2.3   r1 = atomic_load(x, RLX); ❹
1.3  atomic_store(x, data, RLX); ❷     2.4  }
    }                                       }
   }
```
assert($r0 == 2 \implies r1 \neq 0$)

Figure 5.3: A modification to our minimal example in Fig. 5.1, where we including a division by an input argument (dynamically set to one) before writing to `x`.

Cycle:	1	2	3	···	33	34	35
1.1	`st y 1 REL`	`st y 2 REL`					
1.2	div$_{i=1}$	div$_{i=2}$					
1.2		div$_{i=1}$	div$_{i=2}$				
1.2			div$_{i=1}$				
⋮							
1.2					div$_{i=1}$	div$_{i=2}$	
1.3						`st x 1 RLX`	`st x 2 RLX`

Figure 5.4: Pipeline schedule generated by LegUp 5.1 for Fig. 5.3, which violates the program assertion between the 3rd and the 34th cycle.

program:

- if `T1` executes on the first cycle, `r0` will be 0 and `r1` will be 0;
- if `T1` executes on the second cycle, `r0` will be 1 and `r1` will be 0;
- and, if `T1` executes on the third cycle, `r0` will be 2 and `r1` will be 2.

However, this correctness is fragile and can be easily stressed to break.

Consider a second program, in Fig. 5.3, where we pass an argument to `T0`, which we dynamically set to one, and perform a division of the loop index with this argument before writing the result to `x`. This simple change can caused the program assertion to fail on LegUp. Fig. 5.4 shows the pipeline schedule generated by LegUp for `T0` of this second program, where `T0` is provided with a divider with latency of 33 cycles whose execution is pipelined (a new division can start on every cycle). Hence, the division of each loop iteration can start consecutively and their results are ready for the stores at the 34th and 35th cycle respectively.

As in the original example, the aliasing memory operations across iterations are guaranteed to happen in-order. However, the tool can only schedule the two writes to `x` at the 34th and 35th cycle respectively. Independently, the tool also discovers that the second

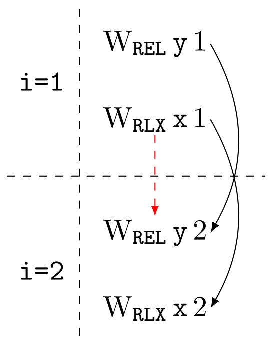


Figure 5.5: The memory events of the both iterations (`i=1` and `i=2`) of `T0` in Fig. 5.1 and 5.3, annotated with the aliasing inter-iteration edges (black solid arrows) and also the additional inter-iteration edge (red dotted arrow) required to ensure correct execution of atomics.

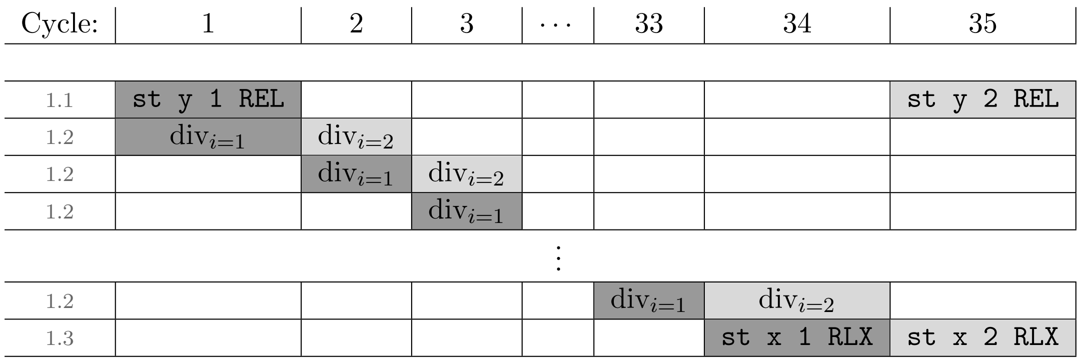
iteration's store of `y` does not depend on the first iteration's store to `x`. As a result, the second iteration's store of `y` is allowed to execute in the second cycle. Unfortunately, this reordering means that between the third and 34th cycle, `T1` will observe that `r0` is two but `r1` is zero, violating the program assertion. This example shows why certain reorderings across iterations must be disallowed to ensure the correctness of programs with atomics. In the next section, we present methods to ensure these types of illegal behaviours are prohibited.

5.4 Extending our methods to support loop pipelining

In this section, we present methods to extend both our thread-local and global analyses from Chapters 3 and 4 to support loop-pipelining. We discuss how to generate the necessary *inter-iteration* memory constraints to complement our existing *intra-iteration* memory constraints to support atomics correctly in the context of loop pipelining.

Our method by example To build an intuition on how to extend our analysis, let's revisit our motivating example in Fig. 5.1. Previously, we discussed how fragile its correctness is when memory scheduling is based on *mem-pipe* of (5.1) from page 110.

Fig. 5.5 shows the memory events of `T0` in Fig. 5.3. We have four memory events (two per iteration) and two iterations (`i=1` and `i=2`). Memory events are discussed thoroughly in §2.3.3.3. As discussed in §5.2, HLS tools preserve aliasing memory constraints across iterations via *inter-iter*, as given by the black solid arrows in Fig. 5.5. However, we also discussed that these constraints are insufficient to ensure correct behaviour of atomics since to first iteration's store of `x` can be reordered with the second iteration's store of `y`. Hence, to ensure this wrong behaviour does not occur, we must inject an additional *inter-iteration* constraint between these two memory events, as shown by the red dotted



Cycle:	1	2	3	···	33	34	35
1.1	`st y 1 REL`						`st y 2 REL`
1.2	$\text{div}_{i=1}$	$\text{div}_{i=2}$					
1.2		$\text{div}_{i=1}$	$\text{div}_{i=2}$				
1.2			$\text{div}_{i=1}$				
⋮							
1.2					$\text{div}_{i=1}$	$\text{div}_{i=2}$	
1.3						`st x 1 RLX`	`st x 2 RLX`

Figure 5.6: The corrected pipeline schedule of Fig. 5.3 after injecting an inter-iteration constraint between the first iteration's write to `x` and the second iteration's write to `y`.

arrow in Fig. 5.5.

Considering memory constraints between two consecutive iterations We only need to consider inter-iteration constraints between two consecutive iterations since these constraints induce a chaining effect across subsequent iterations in the pipeline. For example, one way of expressing the red dotted arrows in Fig. 5.1 is to enforce a rule that all memory operations in the current iteration must be executed before any atomic stores in the next iteration (dependence distance of one). Hence, if we were to execute three iterations of `T0` in Fig. 5.1, instead of two, this rule manifests itself as two constraints: an edge from the first iteration's store of `y` to the second iteration's store of `x` and an edge from the second iteration's store of `y` to the third iteration's store of `x`.

Assumptions Throughout this chapter, we assume two properties on our inputs:

- We assume that our loops only have loop bodies with straight-line code with all control-flow converted via if-conversion;
- and, we assume all accesses to the same C array as aliasing accesses, regardless of the index/offset values.

In the rest of this section, we present extensions to support loop pipelining of atomics, for our thread-local (§5.4.1) and global (§5.4.2) analyses respectively.

5.4.1 Locally-analysing atomics for loop pipelining

In §3.5, we presented two memory models that support the synthesis of SC and weak atomics respectively, in the absence of loop pipelining. In this section, we extend these memory models to support atomics correctly in the context of loop pipelining. Our starting point is *mem-pipe* of (5.1) from page 110, then we enlarge it based on intra-iteration memory constraints of our non-pipelined memory models in §3.5 and, finally, we enforce several additional inter-iteration scheduling rules to support SC (§5.4.1.1) and weak atomics (§5.4.1.2) respectively.

5.4.1.1 Pipelining SC atomics

In §3.5.2, we defined a memory model, *mem-sc* of (3.4) from page 70, that supports SC atomics in a non-pipelined setup. We utilise two intra-iteration rules from this definition ($at\not\uparrow$, $at\not\downarrow$) and two additional inter-iteration rules to enlarge *mem-pipe* to support loop pipelining of SC atomics, as given below:

$$\begin{aligned} \mathit{mem\text{-}sc\text{-}pipe} = {} & \mathit{mem\text{-}pipe} \cup at\not\uparrow \cup at\not\downarrow \cup \\ & at\not\uparrow\text{-}\mathit{inter\text{-}iter} \cup at\not\downarrow\text{-}\mathit{inter\text{-}iter} \end{aligned} \tag{5.2}$$

where

$$\begin{aligned} at\not\uparrow\text{-}\mathit{inter\text{-}iter} &= \{(v, v', 1) \mid v \in V_{\mathrm{at}} \wedge v' \in V_{\mathrm{mem}}\} \\ at\not\downarrow\text{-}\mathit{inter\text{-}iter} &= \{(v, v', 1) \mid v \in V_{\mathrm{mem}} \wedge v' \in V_{\mathrm{at}}\}. \end{aligned}$$

$at\not\uparrow$*-inter-iter* defines that for every atomic operation v in the current iteration and every memory operation v' in the next iteration, there must exist an inter-iteration edge from v to v' with a dependence distance of one. Conversely, $at\not\downarrow$*-inter-iter* defines that for every memory operation v in the current iteration and every atomic operation v' in the next iteration, there must exist an inter-iteration edge from v to v' with a dependence distance of one. These two rules guarantees that all atomics are executed in iteration order and $at\not\uparrow$ and $at\not\downarrow$ ensure that atomics execute in program order within an iteration. Hence, these four rules, together with the aliasing constraints of *intra-iter* and *inter-iter* from *mem-pipe*, guarantee that SC atomics can be executed correctly within a pipelined schedule.

5.4.1.2 Pipelining weak atomics

In §3.5.3, we define a memory model, *mem-weak* of (3.5) from page 71, that supports weak atomics in a non-pipelined context. We utilise five intra-iteration rules from *mem-weak* ($sc\not\uparrow$, $sc\not\downarrow$, $acq\not\downarrow$, $rel\not\uparrow$ and *rar*) and five additional inter-iteration rules to enlarge *mem-pipe* to support weak atomics in the context of loop pipelining, as given below:

$$\begin{aligned} \mathit{mem\text{-}weak\text{-}pipe} = {} & \mathit{mem\text{-}pipe} \cup sc\not\downarrow \cup sc\not\uparrow \cup \\ & acq\not\downarrow \cup rel\not\uparrow \cup rar \cup \\ & sc\not\uparrow\text{-}\mathit{inter\text{-}iter} \cup sc\not\downarrow\text{-}\mathit{inter\text{-}iter} \cup \\ & \mathit{acq\text{-}inter\text{-}iter} \cup \mathit{rel\text{-}inter\text{-}iter} \cup \mathit{rar\text{-}inter\text{-}iter} \end{aligned} \tag{5.3}$$

where

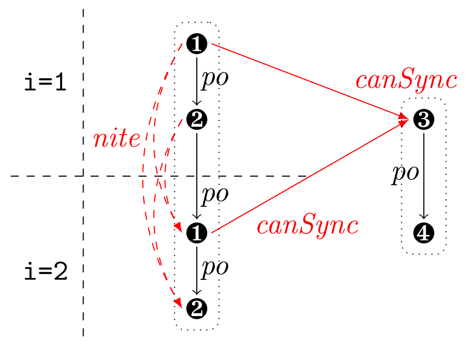
$$\begin{aligned}
\mathit{sc}\!\not\uparrow\text{-}\mathit{inter}\text{-}\mathit{iter} &= \{(v, v', 1) \mid v \in V_{\mathrm{sc}} \land v' \in V_{\mathrm{mem}}\}\\
\mathit{acq}\text{-}\mathit{inter}\text{-}\mathit{iter} &= \{(v, v', 1) \mid v \in V_{\mathrm{acq}} \land v' \in V_{\mathrm{mem}}\}\\
\mathit{sc}\!\not\downarrow\text{-}\mathit{inter}\text{-}\mathit{iter} &= \{(v, v', 1) \mid v \in V_{\mathrm{mem}} \land v' \in V_{\mathrm{sc}}\}\\
\mathit{rel}\text{-}\mathit{inter}\text{-}\mathit{iter} &= \{(v, v', 1) \mid v \in V_{\mathrm{mem}} \land v' \in V_{\mathrm{rel}}\}\\
\mathit{rar}\text{-}\mathit{inter}\text{-}\mathit{iter} &= \{(v, v', 1) \mid v \in (V_{\mathrm{at}} \cap V_{\mathrm{ld}}) \land v' \in (V_{\mathrm{at}} \cap V_{\mathrm{ld}})\\
&\qquad\qquad\qquad\qquad (v, v') \in \mathit{sloc}\}.
\end{aligned}$$

$\mathit{sc}\!\not\uparrow$*-inter-iter* and *acq-inter-iter* are similar rules to $\mathit{at}\!\not\uparrow$*-inter-iter* but apply only to SC and acquire atomics respectively, where for every memory operation v in the current iteration that is either SC or acquire atomic and every memory operation v' in the next iteration, there must exist an inter-iteration edge from v to v' with dependence distance of one. These two rules ensure that all SC or acquire atomics in the current iteration must be executed before any memory operations from the next iteration are executed. $\mathit{sc}\!\not\downarrow$*-inter-iter* and *rel-inter-iter* are similar rules to $\mathit{at}\!\not\downarrow$*-inter-iter* but apply only to SC and release atomics, where for every memory operation v in the current iteration and every SC or release atomic v' in the next iteration, there must exist an inter-iteration edge from v to v' with dependence distance of one. These two rules ensure that all memory operations from the previous iteration are executed before any SC or release atomics in the current iteration can be executed. Finally, *rar-inter-iter* applies to all atomics including relaxed atomics, where for every atomic load v in the current iteration and every atomic load v' in the next iteration to the same location (*sloc*), there must exist an inter-iteration edge from v to v' with dependence distance of one. This rule enforces that all RAR dependencies of atomics are enforced across iterations.

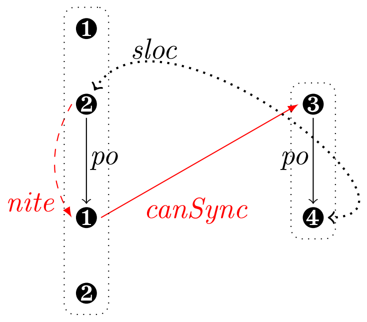
5.4.2 Globally-analysing atomics for loop pipelining

In Chapter 4, we presented a global analysis that consider memory operations of all threads to generate memory constraints for individual threads of a concurrent program. This analysis also assumes the absence of loop pipelining and hence only considers operations of a single iteration of each loop in each thread as inputs. In the context of loop pipelining, however, loop iterations can overlap and execute simultaneously. Therefore, considering a single iteration of each loop in each thread is insufficient to guarantee correctness. Since our work focusses on preserve orderings between two consecutive iterations, we must extend our inputs to consider operations of two consecutive iterations of each loop in each thread. Now, we describe the additional steps required to support loop pipelining for our global analysis by re-visiting our motivating example and then providing a generalisation.

Consider our motivating example in Fig. 5.1. In `T0`, we have a loop with two iterations and we have two memory operations per iteration, labelled as ❶ and ❷ respectively. In `T1`,



(a) Extending inputs and including *nite*



(b) Path enumeration on new inputs

Figure 5.7: Extending global analysis to support loop pipelining for our example in Fig. 5.1.

we have straight-line code with two memory accesses, ❸ and ❹. Before we can globally-analyse this program for loop pipelining, we must expand our inputs to consider memory operations of two consecutive iterations for the loop in `T0`. Then, we extend the *po* relation to the next iteration *i.e.* we include a *po* edge from the first iteration's ❷ to the second iteration's ❶, as seen in Fig. 5.7(a). Next, we also introduce a new relation, the *nite* relation. *nite* relates all memory events in the current iteration to all memory events in the next iteration, as shown by the dashed red arrows labelled as *nite* in Fig. 5.7(a). *nite* is important to keep track of which *po* relations are inter-iteration, to distinguish its appropriate dependence distance.

Based on these new inputs, we analyse the *canSync* edges across threads, as given by the solid red arrows from `T0` and `T1`. Notice that there can also be *canSync* edges from the second iteration of `T0` to `T1`, so our method is exhaustively considering all possible inter-thread synchronisations, even for ones across loop iterations. Then, we perform path enumeration and find that there is only one path that fits the properties described in §4.3.1, as shown in Fig. 5.7(b). Finally, we extract the necessary *ppo* edges from this path. Notice that one of *po* edge we must preserve in `T0` is an inter-iteration edge because this *po* edge is also related by *nite*. Without *nite*, we cannot distinguish between intra- and inter-iteration dependence. Based on this edge, we must include the following SDC constraint within `T0` for the following memory operations: (❶,❷,1). This inter-iteration edge together with the aliasing inter-iteration edges, which global analysis identifies, (❶,❶,1) and (❷,❷,1), ensure that we observe correct program behaviour when the loop is pipelined.

General steps to support loop pipelining In general, we require five additional steps to support loop pipelining for global analysis. Firstly, we must extend the inputs of our global analysis to consider memory operations of two consecutive iterations. Secondly, we must extend *po* and introduce *nite* to relate all memory operations in the current iteration to all memory operations in the next iteration. Thirdly, these new inputs must be considered in its entirety when generating the *canSync* set. Fourthly, these new input and *canSync* sets must be considered for path enumeration based on the same rules discussed in §4.3.1. Finally, after all paths have been enumerated, we extract all the *ppo* edges within these paths. The resulting *ppo* edges can either be intra-iteration or inter-iteration, and we use *nite* to distinguish between them and use the appropriate dependence distance.

Formally, we re-define *ppo* in (4.7) to be:

$$\mathit{ppo} = \mathit{ppo\text{-}intra} \cup \mathit{ppo\text{-}inter} \tag{5.4}$$

where

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

ppo-intra represents the intra-iteration *po* edges within all enumerated paths, where these edges exist in *po* but do not exist in *nite* and hence both v and v' belong to the same iteration, requiring dependence distance of zero. *ppo-inter* represents the inter-iteration *po* edges within enumerated paths, where these edges exist in *po* and also *nite* and hence v belongs to the current iteration and v' belongs to the next iteration, requiring a dependence distance of one.

In §4.4, we proposed an optimisation to alleviate the exponential growth of the number of paths enumerated. Since our five additional steps to support loop-pipelining does not affect the path enumeration step itself, no modifications are required for our optimisation to support loop pipelining for atomics. Understanding that our optimisation works seamlessly even in the context of loop pipelining is critical. When we consider two consecutive iterations for each loop in the program, in practice, it can result in up to four times as many *canSync* edges that are overlapping *canSync* edges, which directly affects the scalability of our global analysis.

5.5 Re-visiting our running example

Here, we re-visit our running example of both thread-local (§3.4) and global (§4.3.3) analyses without loop pipelining. In this chapter, we focus on memory scheduling the same set of memory accesses, but we execute them within a two-iteration for-loop. Consider the program in Fig. 5.8, where in `T0` we have a two-iteration for-loop with the same four memory accesses as the previous running examples . Additionally, we have two more threads, `T1` and `T2` that both have two memory accesses each, which global analysis considers when scheduling `T0`, just as in §4.3.3. `T0` passes messages to `T1` and `T2` via

```
              int a=0,x=0; atomic_int b=0,y=0;
T0                      || T1                   || T2
for(i=1;i<3;i++){       || ❺int r1=ald(&b);     || ❼int r3=ald(&y);
 ❶ a=i;                 ||   if(r1!=0)          ||   if(r3!=0)
 ❷ ast(&b,i,REL);       || ❻  int r2=a;         || ❽  int r4=x;
 ❸ x=i;                 ||                      ||
 ❹ ast(&y,i,REL);       ||                      ||
}                       ||                      ||
                  assert(r2≥r1 && r4≥r3)
```

Figure 5.8: A sample program where T0 sends messages to T1 and T2 from within a two-iteration loop.

Cycle:	1	2	3	4	5	6	7	8
`a=42;`	st_{na} a 1				st_{na} a 2			
`st(&b,1,REL);`		st_{REL} b 1				st_{REL} b 2		
`x=17;`			st_{na} x 1				st_{na} x 2	
`st(&y,1,REL);`				st_{REL} y 1				st_{REL} y 2

Figure 5.9: Pipeline schedule of our running example with thread-local analysis and considering all atomic accesses as SC atomics (the different shades represent different iterations).

independent message passing channels (a pair of atomic and non-atomic locations), by writing to the non-atomic location first and then the atomic location (❶ then ❷ and ❸ then ❹). T1 and T2 receive a message by reading from their atomic location and then their non-atomic location (❺ then ❻ and ❼ then ❽). Since we must always write to the non-atomic location before writing to the atomic location, the value loaded from the atomic location cannot be larger than the value loaded from the non-atomic location, as enforced by the program assertion.

In the next two subsections, we discuss how our loop pipelining extensions of our thread-local and global analyses schedule T0. For all pipelined schedule, its iteration schedule and latency is exactly the same as the non-pipelined case, since the intra-iteration constraints do not change. The key to performance improvements is how our inter-iteration constraints affects the *II*, which permits the overlapping of iterations.

5.5.1 Thread-local analysis with loop pipelining

Fig. 5.9 shows T0's pipeline schedule for our thread-local analysis of SC atomics. Its iteration schedule and latency is the same as in Fig. 3.9. However, our inter-iteration constraints of *mem-sc-pipe* of (5.2) from 115 disallows any loop overlapping. In particular, *at‡-inter-iter* does not allow the second iteration's non-atomic store to a execute before the first iteration's atomic store to y. Consequently, this loop requires eight cycles to execute regardless of loop pipelining, since the initiation interval and loop latency is the

Cycle:	1	2	3	4	5
`a=42;`	st$_{\texttt{na}}$ a 1		st$_{\texttt{na}}$ a 2		
`st(&b,1,REL);`		st$_{\texttt{REL}}$ b 1		st$_{\texttt{REL}}$ b 2	
`x=17;`	st$_{\texttt{na}}$ x 1		st$_{\texttt{na}}$ x 2		
`st(&y,1,REL);`			st$_{\texttt{REL}}$ y 1		st$_{\texttt{REL}}$ y 2

Figure 5.10: Pipeline schedule of our running example with thread-local analysis and considering the consistency modes of each atomic access (the different shades represent different iterations).

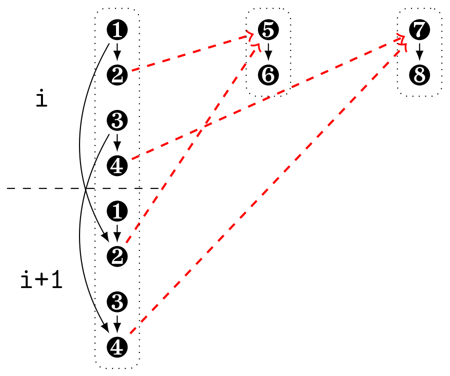


Figure 5.11: The new input set for loop pipelining, where we extend `T0` to consider two consecutive iterations and generate the *canSync* set based on two consecutive iterations (red dashed arrows). The black solid arrows are the preserved *ppo* edges (we omit the aliasing inter-iteration edges for clarity).

same.

Fig. 5.10 shows `T0`'s pipeline schedule for our thread-local analysis of weak atomics. Again, the iteration latency schedule is the same as in Fig. 3.11. However, this time, our inter-iteration constraints of *mem-pipe* of (5.3) from 115 permits iteration overlapping. The critical restriction is *rel-inter-iter* enforces that the first iteration's release store of `y` must execute before the second iteration's store of `b`, as shown by the arrow in Fig. 5.10. Hence, an *II* of two cycles is achievable, compared to its latency of three cycles. Consequently, this loop can complete execution in five cycles, instead of six in the non-pipelined case.

5.5.2 Global analysis with loop pipelining

Fig. 5.11 shows how global analysis treats our running example to support loop pipelining. We extend the inputs of `T0` to consider two consecutive iterations, `i` and `i+1`, and add *po* and *nite* edges between memory accesses in `i` and `i+1`. Then, we analyse *canSync* set based on these new inputs, which spans across both iterations of `T0`, as shown by the red dashed arrows. Next, we enumerated all paths that satisfy properties discussed

Cycle:	1	2	3
❶`a=42;`	$\mathrm{st}_{\mathtt{na}}$ `a` 1	$\mathrm{st}_{\mathtt{na}}$ `a` 2	
❷`ast(&b,1,REL);`		$\mathrm{st}_{\mathtt{REL}}$ `b` 1	$\mathrm{st}_{\mathtt{REL}}$ `b` 2
❸`x=17;`	$\mathrm{st}_{\mathtt{na}}$ `x` 1	$\mathrm{st}_{\mathtt{na}}$ `x` 2	
❹`ast(&y,1,REL);`		$\mathrm{st}_{\mathtt{REL}}$ `y` 1	$\mathrm{st}_{\mathtt{REL}}$ `y` 2

Figure 5.12: Pipeline schedule of global analysis for running example.

in §4.3.1. We see that our pipelining extension generates additional paths consisting of inter-iteration constraints within `T0`. For example, global analysis identifies that there is a path from the `i`-th iteration's ❶ to `i+1`-th iteration's ❷ to ❺ and finally to ❻. This path involves an inter-iteration edge from ❶ to ❷. We identify this edge as inter-iteration, since it is also be related by *nite*.

Fig. 5.12 shows `T0`'s pipeline schedule for our global analysis. We can achieve the same iteration schedule and latency of two cycles, just as in Fig. 4.6. However, the critical behaviour that ensures correctness is that the atomic release store must not happen before its corresponding non-atomic store within each iteration, which does not prevent the loop iterations from overlapping. Therefore, an *II* of one is achievable and this loop only takes three cycles to complete execution, instead of four in the non-pipelined case.

5.6 Ensuring correctness of our loop-pipelining support

As we described earlier in this chapter, the C memory model does not distinguish between memory events of different iterations. Memory events of different iterations are simply related by *po*. Additionally, both our thread-local and global analyses only require distinguishing between memory events of different iterations to employ the appropriate HLS dependence distance. This dependence distance is irrelevant to our event signature and hence does not need to be given special attention. Therefore, the verification of our thread-local and global analyses of Chapter 3 and 4 also holds true for this chapter, without necessitating the any extensions or modifications.

5.7 Implementing loop-pipelining in LegUp 5.1

Fig. 5.13 shows the different stages of LegUp flow involved when implementing loop-pipelining. Modulo scheduling is implemented as a LLVM transformation pass in the `opt` executable. After the source code is compiled by `clang` into LLVM IR, the entire program is allocated and then each thread is given to the modulo scheduler individually within the `opt` executable. Then, LegUp generates intra- and inter-iteration aliasing memory constraints for the consideration of the modulo scheduler. The modulo scheduler is then

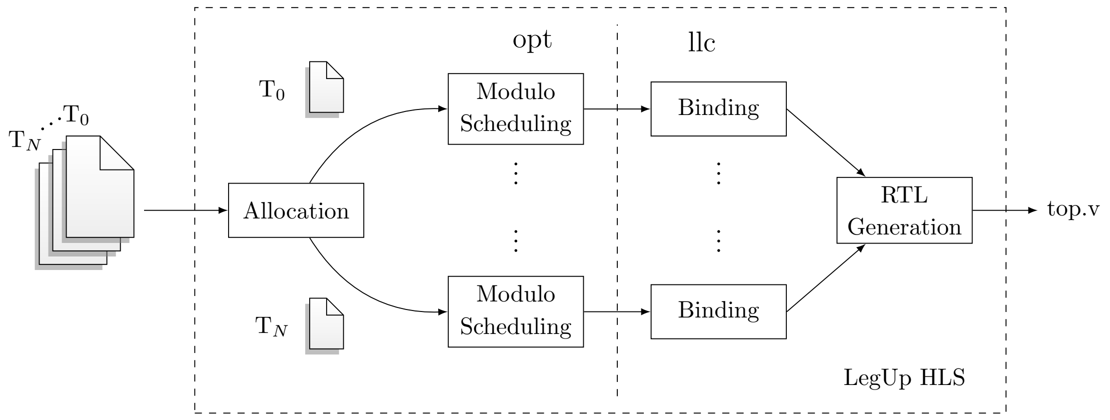


Figure 5.13: Supporting loop-pipelining via LegUp 5.1 HLS tool flow via thread-local analysis, where we inject both our intra- and inter-iteration constraints within the Modulo Scheduling step of each thread.

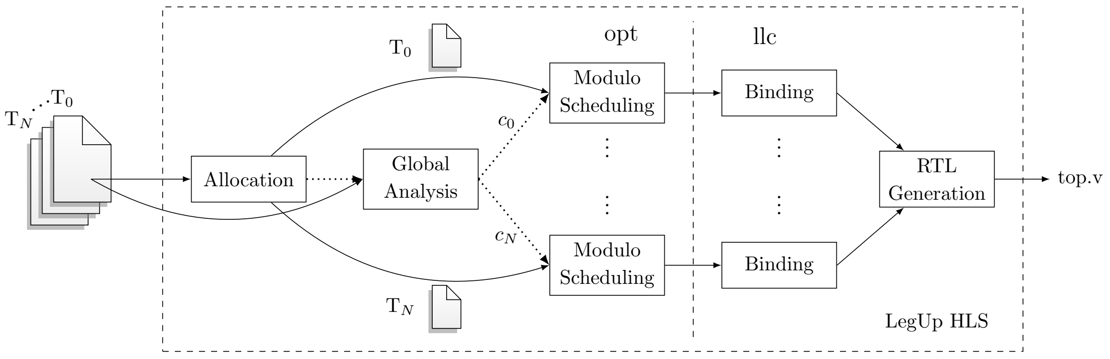


Figure 5.14: Supporting loop-pipelining via LegUp 5.1 HLS tool flow via global analysis, where we perform global analysis and then inject both our intra- and inter-iteration constraints of each thread to their respective Modulo Scheduling steps.

tasked to compute a valid pipeline schedule. If a valid schedule is found, several LLVM metadata are included to the each IR including: 1) the loop latency, 2) the initiation interval, 3) each operation's pipeline stage and 4) each operation's start time within the pipeline stage. Finally, the RTL Generation stage in the `llc` utilises this metadata to generate a pipeline schedule on hardware.

To implement our thread-local analysis of atomics in LegUp, we simply augment the modulo scheduler to inject our additional intra- and inter-iteration memory constraints, as described in §5.4.1. We provide constraints to each thread's modulo scheduler based on *mem-pipe* from (5.2) and *mem-weak-pipe* from (5.3) to support loop pipelining of SC and weak atomics respectively.

Since our global analysis requires an LLVM pass that iterates over all LLVM functions and global variables, we must introducing an additional LLVM pass before modulo scheduling. Fig. 5.14 shows how we implement loop-pipelining via global analysis, where

Table 5.1: Design points for evaluation of Chapter 5.

Short name	Analysis	Weak	Pipelining	Model	Ref.
LocalSC	Local	No	No	*mem-sc*	§3.5.2
LocalWeak	Local	Yes	No	*mem-weak*	§3.5.3
GlobalSC	Global	No	No	*ppo*	§4.3.1
GlobalWeak	Global	Yes	No	*ppo*	§4.3.2
PipelinedLocalSC	Local	No	Yes	*mem-sc-pipe*	§5.4.1.1
PipelinedLocalWeak	Local	Yes	Yes	*mem-weak-pipe*	§5.4.1.2
PipelinedGlobalSC	Global	No	Yes	*ppo-intra*, *ppo-inter*	§5.4.2
PipelinedGlobalWeak	Global	Yes	Yes	*ppo-intra*, *ppo-inter*	§5.4.2

we implement our LLVM pass after the Allocation stage. Then, we provide the necessary intra-iteration and inter-iteration constraints to individual threads, as shown by the dotted arrows labelled as c_0 to c_N. Modulo scheduling of each thread is computed entirely based on the constraints provided by global analysis, since our global analysis is standalone.

5.8 Evaluation

We evaluate our loop pipelining extensions of both thread-local and global analyses on two sets of experiments: memory-dominant and compute-dominant experiments. We described their differences thoroughly in §2.5.3.2 but, in essence, we want understand how loop pipelining benefits programs that are memory-intensive but also compute-intensive. In this section, we discuss the following:

- In §5.8.1, we first describe all the design points for evaluation.
- In §5.8.2, we discuss the runtime performance of our memory-dominant experiments.
- In §5.8.3, we discuss the runtime performance of our compute-dominant experiments.
- Finally, in §5.8.4, we discuss the resource utilisation overheads of loop pipelining.

5.8.1 Design points

In this chapter, we evaluate eight different design points, as summarised in Table 5.1. Four design points are carried forward from Chapters 3 and 4, which are our thread-local and global analyses without loop pipelining: *LocalSC*, *LocalWeak*, *GlobalSC* and *GlobalWeak*.

In this chapter, we treat these four design points as our baseline and then we enable loop pipelining for all four of these design points: *PipelinedLocalSC*, *PipelinedLocalWeak*, *PipelinedGlobalSC* and *PipelinedGlobalWeak*. *PipelinedLocalSC* and *PipelinedLocalWeak* implement loop-pipelining via thread-local analysis, as in §5.4.1. *PipelinedLocalSC* treats all atomic accesses as SC atomics and *PipelinedLocalWeak* is sensitive to weak atomics. *PipelinedGlobalSC* and *PipelinedGlobalWeak* implement loop-pipelining via global analysis, as in §5.4.2. The only difference between *PipelinedGlobalSC* and *PipelinedGlobalWeak* is their *canSync* definitions. The *canSync* definition of *PipelinedGlobalSC* considers all atomics as SC atomics whereas *PipelinedGlobalWeak* supports weak atomics.

5.8.2 Runtime performance of memory-dominant experiments

First, we discuss the runtime performance of our eight design points on memory-dominant experiments. This set of experiments consists of benchmarks transferring data atomically without performing any computation on the data. This setup is best-fitted to understand the benefit of our analysis in isolation. For this reason, we utilised these experiments in previous chapters. Fig. 5.15 shows the runtime performance of our memory-dominant experiments for all eight design points.

5.8.2.1 Loop pipelining generally improves performance

Enabling loop pipelining improves the performance of *LocalWeak*, *GlobalWeak* and *GlobalSC*, but worsens the performance of *LocalSC*. The key to these improvements is the possibility of overlapping iterations.

PipelinedLocalSC* versus *LocalSC In most cases, our thread-local pipelining rules for SC atomics enforces that all atomics in the current iteration must be executed before any operations from the next iteration. Since our benchmarks typically end their loop bodies with an atomic access, it is most likely that our rules prohibit any loop overlappings. Hence, across all our experiments, we see that *PipelinedLocalSC* performs similarly to its non-pipelined counterpart, *LocalSC*. On average, *PipelinedLocalSC* is only 2% faster than *LocalSC* with a maximum of 35%. Additionally, *PipelinedLocalSC* can also be up to 20% slower than *LocalSC* because enabling loop pipelining require additional control signals, forming the critical path that reduces clock frequency.

PipelinedLocalWeak* versus *LocalWeak In contrast, our thread-local pipelining rules of weak atomics are less restrictive compared to SC atomics. Our benchmarks typically begin their loop bodies with an acquire atomic and end with a release atomic, which are subject to fewer constraints than SC atomics. Hence, on average, *PipelinedLocalWeak* is 1.6× faster than *LocalWeak* with a maximum of 3.3×.

Global analysis with pipelining is better When loop pipelining is enabled, global analysis always identifies that routines across iterations can be overlapped to a certain extent without breaking program correctness. Hence, we see that, on average, *PipelinedGlobalSC* is 1.4× faster than its non-pipelined counterpart *GlobalSC* with a maximum of 1.9×. We also see that, on average, *PipelinedGlobalWeak* is 1.4× faster than the *GlobalWeak* with a maximum of 2.5×.

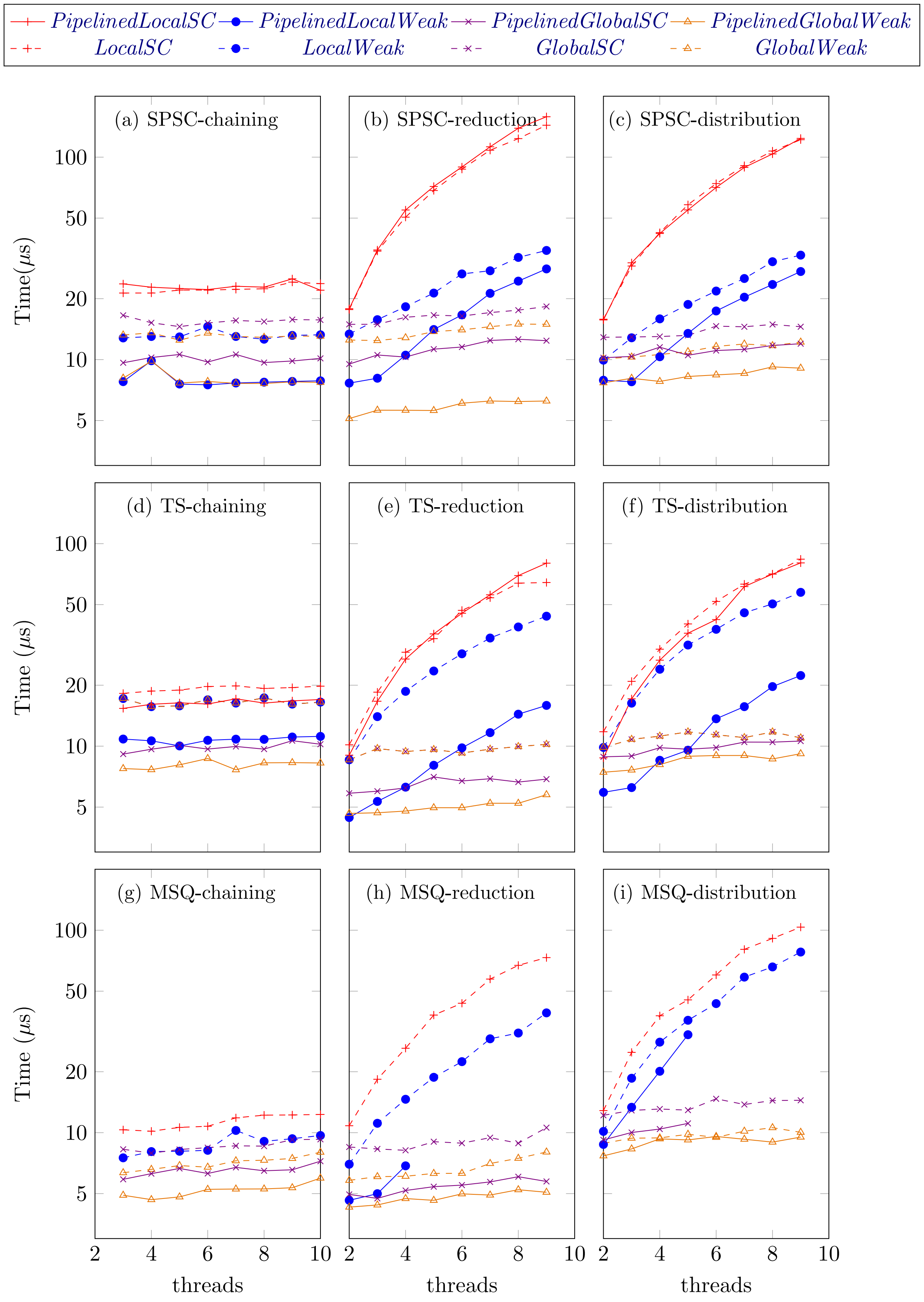


Figure 5.15: Runtime performance of all our design points in Table 5.1 on our memory-dominant experiments.

5.8.2.2 Both global analysis and weak atomics maintain their influences

In the absence of loop pipelining, both global analysis and weak atomics individually impacts performance and their impacts are similar when loop pipelining is enabled.

Discussing global analysis In the absence of loop pipelining, global analysis identifies parallelism within and across routines. In the context of loop pipelining, global analysis also identify parallelism across iteration of routines. Hence, we see that, on average, *PipelinedGlobalSC* and *PipelinedGlobalWeak* is 3× and 1.4× faster than *PipelinedLocalSC* and *PipelinedLocalWeak* respectively, with a maximum of 13× and 4.5×.

Discussing weak atomics Furthermore, weak atomics has a better influence on performance when loop pipelining is enabled. In the context of loop pipelining, weak atomics increases the scope of reorderings to consider operations of different iterations. We see that, on average, *PipelinedLocalWeak* is 3.4× faster than *PipelinedLocalSC* with a maximum of 5.7×. In contrast, without loop pipelining, *LocalWeak* is only 1.6× faster than *LocalSC* with a maximum of 4×. Also, we see that, on average, *PipelinedGlobalWeak* is 1.3× faster than the *PipelinedGlobalSC* with a maximum of 2×. In contrast, without loop pipelining, *GlobalWeak* is 1.2× faster than *GlobalSC* with a maximum of 1.5×.

5.8.2.3 Limitations of modulo scheduling

We also see that there is trend between the number of constraints and the likelihood of LegUp's modulo scheduler generating a pipeline schedule. Larger and more complex benchmarks lead to higher RAM usage during modulo scheduling. This trend is most evident for our queue experiments, as we run out of RAM quickly (6GB of RAM) depending on design points. Each design point is required to generate 24 data points across three graphs in Figures 5.15(g), 5.15(h) and 5.15(i). However, due to the difference in the number of constraints per design point, we see that *PipelinedLocalSC* cannot generate any data points, *PipelinedLocalWeak* generates only 4 data points, *PipelinedGlobalSC* generates only 20 data points and *PipelinedGlobalWeak* can generate all data points. Our experiments show that the use of global analysis and weak atomics not only improves runtime performance, but also improves the likelihood of generating a pipelined schedule without exhausting memory.

5.8.3 Runtime performance of compute-dominant experiments

Next, we discuss the runtime performance of our eight points for our compute-dominant experiments. We thoroughly discuss compute-dominant experiments in §2.5.3.2. This set of experiments include long-latency division, making the workloads heavy on computation rather than memory accesses, which is useful understand how our pipeline scheduling is

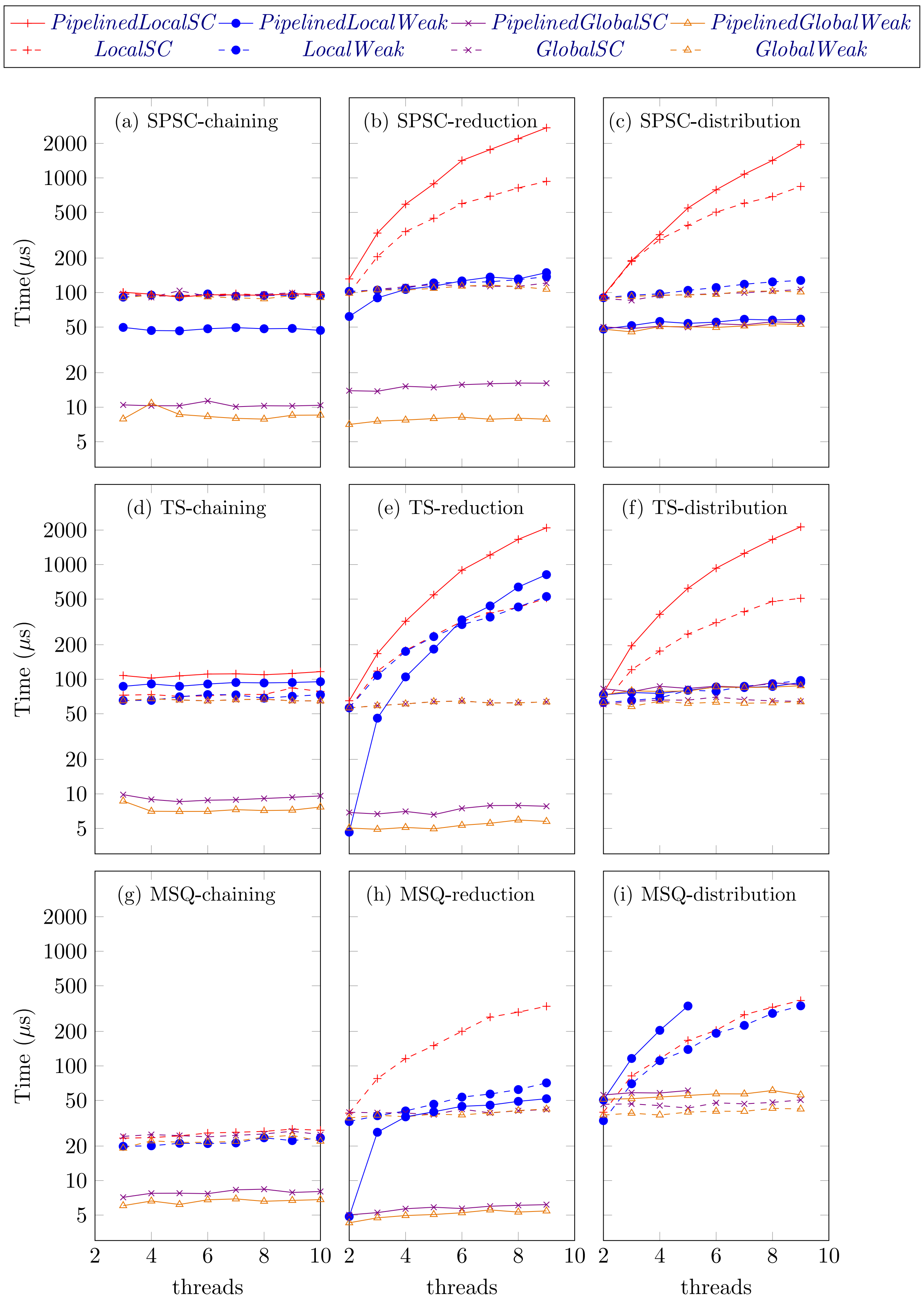


Figure 5.16: Runtime performance of our design points in Table 5.1 on our compute-dominant experiments.

affected by computation. In particular, we shall see that *II* can be influenced by these divisions. Fig. 5.16 shows the runtime performance of our compute-dominant experiments.

5.8.3.1 Speedups of thread-local analysis varies.

PipelinedLocalSC* versus *LocalSC The performance of *PipelinedLocalSC* is worse than our memory-dominant experiments. The *II* of *PipelinedLocalSC* is similar to its latency, just as in our memory-dominant experiments. However, compute-dominant experiments have much larger latencies and it directly influences the *II*s. The larger the *II*, the larger the circuit area and the lower the clock frequencies. So, we obtain the worst-case of no *II* improvements and lowered frequencies for *PipelinedLocalSC*. When the *II* is small enough, this effect is contained, as we can see for all chaining experiments and for experiments with lower thread counts. However, on average, *PipelinedLocalSC* is 1.5× slower than *LocalSC* with a maximum of 4.1×.

PipelinedLocalWeak* versus *LocalWeak In contrast, thread-local analysis of weak atomics provides speedups when enabling loop pipelining. On average, *PipelinedLocalWeak* is 1.5× faster than *LocalWeak* with a maximum of 12×. However, the different data-flow patterns produce different trends.

Chaining experiments For the buffer experiments, on average, *PipelinedLocalWeak* is 2× than *LocalWeak*, as seen in Fig. 5.16(a). However, for the stack experiments, *Pipelined-LocalWeak* is about 30% slower than *LocalWeak*, as seen in Fig. 5.16(d). This is because the stack's routines are small and only consist of two different memory locations per data structure. Hence there are many aliasing constraints which our analysis cannot avoid.

Reduction experiments The *II* on *PipelinedLocalWeak* is typically half its latency, resulting in *PipelinedLocalWeak* producing half the cycle count of *LocalWeak*. However, as the thread count grows, the latency of *PipelinedLocalWeak* also grows, which means the *II* is also growing. Again, larger *II*s lead to lower frequencies. Hence, for higher thread counts, the clock frequency penalties of *PipelinedLocalWeak* is more evident. In particular, we see in Fig. 5.16(e) that *PipelinedLocalWeak* is 12× faster than *LocalWeak* at two threads but becomes slower than *LocalWeak* by up to 2.4× at nine threads.

Distribution experiments Distribution experiments have a similar trend to the chaining experiments. For the buffer experiments, *PipelinedLocalWeak* is 2× faster than *Lo-calWeak*, as seen in Fig. 5.16(c). For the stack experiments, *PipelinedLocalWeak* perform similarly to *LocalWeak* since stack routine is small and consists of many aliasing constraints, as seen in Fig. 5.16(f). For the queue experiments, *PipelinedLocalWeak* is 1.8× slower than *LocalWeak*,, as seen in Fig. 5.16(i). Although their cycle counts are similar, the former suffers from larger frequency penalties.

5.8.3.2 Speedups of global analysis are consistently better.

Global analysis consistently produces better speedups for compute-dominant experiments, compared to memory-dominant experiments, since the IIs generated by global analysis is typically unaffected by the loop latency. Global analysis produces similar IIs for both memory- and compute-dominant experiments, even though divisions increase the iteration latency of most compute-dominant threads. Hence, the global analysis speedups of compute-dominant experiments is far better than memory-dominant experiments. On average, *PipelinedGlobalSC* is 4.8× faster than *GlobalSC*, with a maximum of 10×. In contrast, *PipelinedGlobalSC* is only 1.4× faster than *GlobalSC*, with a maximum of 1.9× for memory-dominant experiments. Additionally, on average, *PipelinedGlobalWeak* is 5.7× faster than *GlobalWeak*, with a maximum of 15×. In contrast, *PipelinedGlobalWeak* is only 1.4× faster than *GlobalWeak*, with a maximum of 2.5× for memory-dominant experiments.

In very few cases, global analysis can be slower after enabling loop pipelining, which is seen only with the distribution experiments, in Figures 5.16(f) and 5.16(i). The memory access patterns of distribution prohibits iteration overlapping for both thread-local and global analysis. Therefore, the cycle counts of the pipelined and non-pipelined circuits are similar but the clock frequencies of the pipelined circuits are lower.

5.8.3.3 Missing data points

Even for the compute-dominant experiments, Legup's modulo scheduler fails to generate a pipeline schedule for some queue experiments. None of the *PipelinedLocalSC* could be generated. Only 12 out of the 24 data points could be generated for *PipelinedLocalWeak*. On the other hand, all data points for *PipelinedGlobalSC* and *PipelinedGlobalWeak* were generated. This trend re-affirms that the queue experiments produce the largest number of scheduling constraints, making the pipeline computation exponentially harder.

5.8.4 Resource utilisation

Fig. 5.17 shows the relative resource utilisation overheads when we enable loop pipelining for both memory- and compute-dominant experiments. In most cases, loop pipelining requires both additional LUTs and registers for all design points on both sets of experiments. On average, memory-dominant experiments require additional 15% LUTs and 5% registers and compute-dominant experiments require additional 11% LUTs and 8% register to implement loop pipelining.

LUT usage relies on the II. The larger the II, the larger the circuit area required to implement loop pipelining. Hence, we see that enabling loop pipelining for *PipelinedLocalSC* consumes most relative LUTs. This is because *PipelinedLocalSC* struggles to overlap

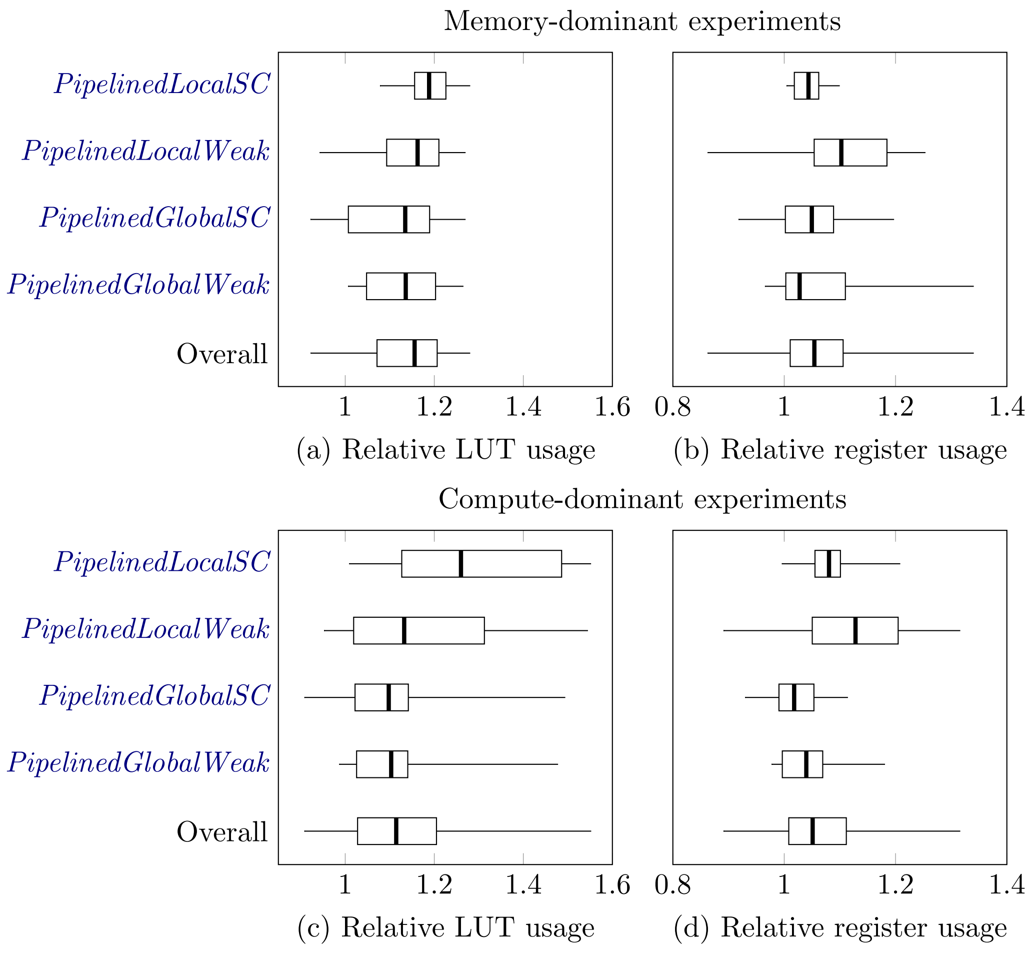


Figure 5.17: Relative resource utilisation overheads when enabling loop pipelining for both our memory- and compute-dominant experiments.

loop iterations and therefore its *II* is large and similar to its loop latency. In contrast, *PipelinedLocalWeak* require fewer LUTs to implement loop pipelining, since this design point typically achieves *II*s that are half the loop latency. Finally, *PipelinedGlobalSC* and *PipelinedGlobalWeak* require the smallest LUT usage to implement loop pipelining, since their *II*s are often unaffected by the loop latency.

Furthermore, since the compute-dominant experiments have large latencies, its relative LUT variations are higher than the memory-dominant experiments. The third quartiles and maximums of *PipelinedLocalSC* and *PipelinedLocalWeak* are larger for compute-dominant experiments in Fig. 5.17(c), compared to memory-dominant experiments in Fig. 5.17(a). Sometimes loop pipelining can results in the usage of fewer LUTs, however these cases are rare and often due to the variabilities in synthesis optimisation.

The relative register usage when implementing loop pipelining varies for different design points and experiments, since both LegUp 5.1 and Quartus implement register duplication when deemed fit.

$$\begin{tikzcd}
\mathit{GlobalWeak} \arrow[r, "1.4\times"] & \mathit{PipelinedGlobalWeak} \\
\mathit{GlobalSC} \arrow[r, "1.4\times"] & \mathit{PipelinedGlobalSC} \\
\mathit{LocalWeak} \arrow[r, "1.6\times"] & \mathit{PipelinedLocalWeak} \\
\mathit{LocalSC} \arrow[r, "1.0\times"] & \mathit{PipelinedLocalSC}
\end{tikzcd}$$

(a) memory-dominant experiments

$$\begin{tikzcd}
\mathit{GlobalWeak} \arrow[r, "5.7\times"] & \mathit{PipelinedGlobalWeak} \\
\mathit{GlobalSC} \arrow[r, "4.8\times"] & \mathit{PipelinedGlobalSC} \\
\mathit{LocalWeak} \arrow[r, "1.5\times"] & \mathit{PipelinedLocalWeak} \\
\mathit{LocalSC} \arrow[r, "0.7\times"] & \mathit{PipelinedLocalSC}
\end{tikzcd}$$

(b) compute-dominant experiments

Figure 5.18: Average speedups of design points in Table 5.1 for each set of experiments.

5.9 Conclusion

In this chapter, we extend our thread-local and global analyses from Chapter 3 and 4 to support loop pipelining of fine-grained C concurrency. Our method is general applicable to any programs with loops consisting of atomics. We do so by complementing our intra-iteration scheduling constraints of Chapters 3 and 4 with the necessary *inter-iteration* constraints to support loop pipelining correctly. For both thread-local and global analyses, we can tailor our methods to be sensitive to weak atomics. We implement our methods in the LegUp 5.1 and evaluate them on two set of experiments that are memory- and compute-dominant respectively.

Fig. 5.18 recaps the key results from our evaluation. Enabling loop pipelining for thread-local analysis of SC atomics (*LocalSC*) does not yield in performance improvements. In contrast, enabling loop pipelining for thread-local analysis of weak atomics (*LocalWeak*) yields speedups under 2× for both memory- and compute-dominant experiments, which shows that this analysis generally produce *II*s that are half the loop latency. On average, loop pipelining of global analysis yields performance of under 2× for memory-dominant experiments, whereas yields performance of around 5× for compute-dominant experiments. This is because the *II*s achieved by global analysis is the same regardless of memory- or compute-dominant experiments. However, the compute-dominant experiments have larger latencies and therefore our speedups are better for those experiments.

In the last two chapters, we presented methods to synthesise fine-grained C concurrency with thread-local and global analyses. In this chapter, we present extensions to support loop pipelining for fine-grained C concurrency for both these analyses. In the next chapter, we present a real-world case study that showcases the benefits of all these analyses.

6. Case study: Google's PageRank

6.1 Introduction

In this chapter, we demonstrate the benefits of our analyses from previous chapters on a real-world example. We transform Google's PageRank algorithm to use lock-free streaming and dynamic load-balancing via fine-grained atomics. PageRank was first devised by Brin and Page [131] to improve web search engines in 1998. Prior to their work, web pages were ranking based on academic citation analysis. Academic citation analysis, typically, performs ranking based on a flat structure which works well for peer-reviewed academic publications. However, this approach is ill-suited for webpage ranking since there is no quality control associated with webpages. The key idea of PageRank is to rank web pages based on the graph of the web, instead of a flat-structure ranking approach [131]. A webpage is highly-ranked if the sum of the ranks of its backlinks (in-edges) is high.

PageRank is interesting as a case study because it is an important, well-known and widely-adopted algorithm and it is a key component of Google's search engine infrastructure [132]. Hence, it attracts a lot of hardware acceleration interests from both industry and academia. Active research on accelerating PageRank is being conducted on a range of hardware architectures. In particular, there are several works on accelerating PageRank on GPUs [133, 134, 135, 136, 137].

PageRank has several interesting features that make it a good real-world example to showcase our analyses. Firstly, the PageRank algorithm is in the form of a sparse-matrix vector multiplication (SpMV), which has several implications. This SpMV pattern is applicable to wide range of graph algorithms, which makes it one of the important problems to solve. Also, the SpMV access pattern is non-trivial, which means it is not easily pipelinable or optimised for HLS. It also means that PageRank workload is most unlikely to be partitioned evenly across multiple threads, which makes it an irregular graph application. Secondly, PageRank's computation involves long-latency floating-point operations, such as addition and division. Therefore, PageRank consists of both computation and data transfer as part of the algorithm, which closely represents a real-world program. Finally, PageRank consists of CAS operations. CASes cannot be synthesised by most HLS tools but our work supports the synthesis of CAS operations via LegUp HLS.

Based on these reasons, we transform an HLS implementation of PageRank into an implementation that is pipelined, streamed and load-balanced to improve its performance. We do so by incorporating lock-free streaming and work-stealing on PageRank. Then, we apply our various analyses from previous chapters to synthesise these implementations correctly and efficiently. In this chapter, we discuss the following:

- In §6.2, we discuss the PageRank algorithm in depth and its C implementation that we compile to hardware via LegUp HLS .
- Then, in §6.3, we discuss how to turn this baseline implementation of PageRank into a lock-free streaming implementation.
- Finally, in §6.4, we demonstrate evidence of workload imbalance for our parallelised PageRank implementations and discuss how lock-free work-stealing improves the workload distribution.

6.2 Google's PageRank algorithm

First, we discuss the theory of PageRank in §6.2.1. Then, we discuss a C implementation of PageRank from the Pannotia benchmark suite for GPUs [133], which is inspired by Pregel's description [138], in §6.2.2. Finally, we discuss a hardware realisation of this C implementation in LegUp, which we regard as a baseline, in §6.2.3.

6.2.1 Theory

The simplified version of PageRank is defined as follows:

$$R_{t+1}(u) \leftarrow \frac{1-d}{N} + d \sum_{v \in B_u} \frac{R_t(v)}{N_v} \tag{6.1}$$

Let u be a web page, B_u be the set of backlinks of u and N be the number of webpages in the graph. Each webpage v that points to u contributes to its rank, $R_{t+1}(u)$. We individually divide the rank of each webpage v by the number of forward links it has, N_v. Then we sum these contributions and multiply it by a damping factor d. This result is then add it by $\frac{1-d}{N}$ to achieve the final rank of u. This rank computation of webpages must be executed over multiple iterations to ensure the effects of individual ranks can propagate across the entire graph before converging. d is the damping factor, which is the probability of a random surfer continuing to click on links on the current web page. Experimentally, it is shown that 0.85 is the right value of d for good convergence [132]. Also, typically, each rank $R_0(v)$ is initialised as $\frac{1}{N}$.

PageRank can also be stated in matrix form [131]. Let A be a square matrix with rows and columns corresponding to the individual web pages. Let $A_{u,v} = \frac{1}{N_u}$, if there is an edge from u to v and zero otherwise. Let R be a rank vector over web pages. Then, we

```
           int row[NODES]; int col[EDGES];
float pagerank1[NODES]; atomic_uint pagerank2[NODES];
for(int v = 0; v<NODES; v++) {
 int start = row[v];
 int end = row[v+1];
 for(int edge=start; edge<end; end++) {
  int u = col[edge];
  add_float_atomic(&pagerank2[u],
      pagerank1[v]/(float)(end-start));
 }
}
```

Figure 6.1: Google's Pagerank from Pannotia

have $R = cAR$. Thus, R is an eigenvector of A with eigenvalue c. Also since A is a sparse matrix, PageRank is an SpMV problem and an irregular workload.

6.2.2 C Implementation

Fig. 6.1 shows a C implementation of the PageRank algorithm, which we adapted from Pannotia [133]. Their original implementation was written in OpenCL. We transform their implementation into a concurrent C implementation via pthreads with minimal modifications. The only changes we made to Pannotia implementation was to ensure it fits with the pthreads execution and memory model:

- We use pthreads fork/join functions to parallelise PageRank computations, instead of OpenCL work items within a mutli-dimensional index space [5, §3.2].
- We remove all OpenCL memory scoping of shared memory by placing all shared memory as arrays synthesised on-chip [5, §3.3].

This pthreads implementation is our *baseline* implementation of PageRank. In future sections, we further transform this pthread implementation to improve its performance via the use of lock-free data structures.

Our baseline implementation focuses on computing the summation of the rank contributions, which is the most time-consuming part of PageRank since each rank contribution requires a floating-point division followed by floating-point addition. The performance of the rank contributions are constant across iterations, because the compute and memory access patterns are the same. Consequently, the interesting performance improvements result from speeding up computation within an iteration.

C memory arrays PageRank consists of three input arrays and one output arrays. `pagerank1` is the input rank R_t and `pagerank2` is the output rank R_{t+1} respectively. The `row` and `col` arrays are the input matrix A in the Compressed Sparse Row (CSR)

```
 1 void add_float_atomic(atomic_uint *address, float inc) {
 2  uint oldValUint = *address;
 3  float oldVal = *((float*) &oldValUint);
 4  bool success = false;
 5  while(!success){
 6   float newVal = oldVal + inc;
 7   uint newValUint = *((uint*) &newVal);
 8   success = atomic_compare_exchange_strong( address,
         &oldValUint, newValUint);
 9   oldVal = *((float*) &oldValUint);
10  }
11 }
```

Figure 6.2: Atomic compare-and-swap on a floating-point memory location.

format [139, §11.5.1]. The CSR format reduces the storage space of sparse matrices by only storing non-zero entries. The `row` array contains the starting index at which each node must access the `col` array. The `col` array contains the forward links of each node. In other words, the `row` array holds the `col` addresses that each node must access and hence their sizes are given by the number of nodes and edges of the input graph respectively.

Pagerank's loops The algorithm consists of two loops iterating over each node `v` and each of its forward links (`v`,`u`) respectively (lines 1-4). We represent symbols u and v in (6.1) as `u` and `v` in the C implementation. It is also important to note that (`v`,`u`) is not only a forward link for `v` but also a backward link for `u`. Consequently, notice that this C implementation computes PageRank over all forward links of `v`, rather than all backward links of `u` as defined in (6.1). This is because the forward links of a webpage are known at download time without requiring any additional processing, unlike acquiring its backward links. Hence, we iterate over all forward links of `v` but our rank updates treats them as a backward links of `u`. In practice, this method has one complication. In a parallel implementation, several threads can be updating the same `u` simultaneously, which why atomic additions are required, as seen in line 6 of Fig. 6.1.

Atomic floating-point addition Although the floating-point addition of PageRank must be atomic, the C library does not support atomic accesses for floating-point numbers. This issue is overcome with two decisions. Firstly, the output ranks are declared as atomic unsigned integers, but the contents of these ranks are casted into and from floating-point. Secondly, the addition itself is implemented with a CAS operation within a loop. This ensures that the addition keeps spinning until it is successful, since a non-atomic addition does not guarantee atomicity as the rank can be simultaneously updated by other threads.

Fig. 6.2 shows the implementation of `add_float_atomic` function, which consists of a loop (line 5) spinning until a CAS operation is successful (line 8). The CAS operation

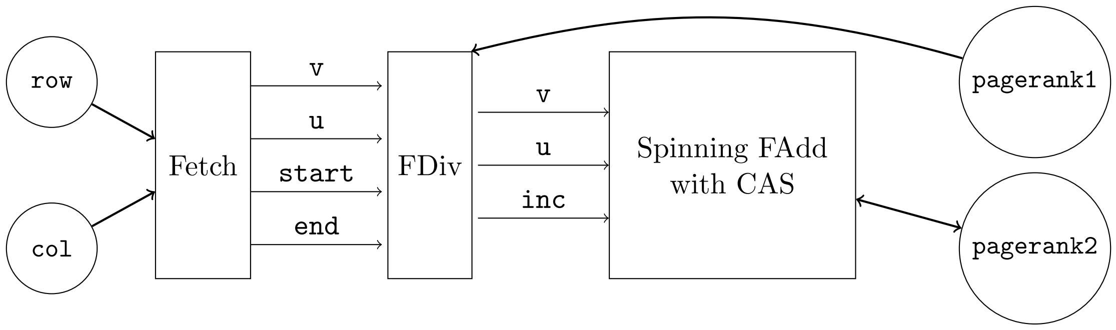


Figure 6.3: Computation steps executed by PageRank, as described in Fig. 6.1, where rectangles are hardware stages and circles are memory arrays.

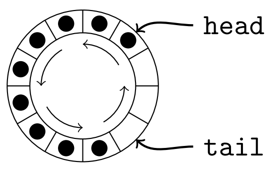
is only successful if the floating-point contents of the pointed `address` is loaded, added by `inc` and swapped by the CAS before any other updates to the same `address` happen. This guarantees the atomicity of the floating-point addition. Lines 6 and 9 show the casting of unsigned integers to floating-point numbers and vice versa to assist with the implementation of a floating-point atomic addition.

6.2.3 Hardware realisation

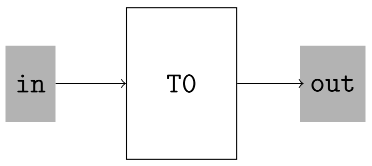
Fig. 6.3 shows various hardware stages (rectangles) and memory arrays (circles) with the memory access patterns (arrows between circles and rectangles) and data-flow between stages (labelled arrows between rectangles and rectangles), when we synthesise this C implementation of PageRank via LegUp HLS.

Hardware stages This baseline PageRank consists of three hardware stages that are encapsulated within a single pthread (software thread). The first stage, `Fetch`, accesses the CSR input matrices from the `row` and `col` arrays, which provides the node and edge list information (line 2, 3 and 5 in Fig. 6.1). This stage generates a unique (`v`,`u`) pair, which propagates through the different hardware stages. The next hardware stage, `FDiv`, perform a floating-point division of the node `u`'s input rank and its number of forward links. The final stage, `Spinning with FAdd with CAS`, performs the atomic floating-point addition, described in Fig. 6.2.

Memory arrays Since we utilise LegUp's pure-hardware flow, all four memory arrays must be synthesised on-chip as block RAMs. This restriction constraints the sizes of our memory arrays, which means it is not likely that we fit an entire graph on-chip. As a workaround, we partition our input graph into a sub-graph that fit on-chip and test our hardware performance based on this sub-graph. Our input graph is the DLBP co-author database from the UCF sparse matrix collection [140] (Pannotia uses the same graph). We partition the co-author graph by picking the first N authors (nodes) and preserving



(a) The circular buffer, diagrammatically.



(b) Streaming example where a thread `T0` reads from `in` and writes to the `out` (both of which are SPSC buffers).

Figure 6.4: Understanding the SPSC buffer and the context in which we use it.

all E relations between these N nodes. We can fit up to $N = 8192$ nodes and $E = 75000$ edges on-chip on a CycloneV FPGA.

6.3 Implementing lock-free streaming of PageRank

In this section, we transform the baseline PageRank implementation into an implementation capable of lock-free streaming. We do so in three parts:

- First, in §6.3.1, we identify an optimal coding template using single-producer-single-consumer (SPSC) buffers to achieve one-to-one lock-free streaming with low initiation intervals (*II*s).
- Then, in §6.3.2, we identify a pipelinable partitioning of the baseline PageRank that utilises our coding template to achieve lock-streaming of PageRank.
- Finally, in §6.3.3, we synthesise this streaming implementation using our various analyses from previous chapters and compare these results against the baseline.

6.3.1 One-to-one lock-free streaming via SPSC buffers

In this subsection, we describe a coding template to achieve high-performance one-to-one streaming between threads using only SPSC buffers.

Recap of SPSC buffer We described SPSC buffers thoroughly in §2.5.1.1. This data structures consists of two atomic pointers, `head` and `tail` pointers, and a non-atomic `array` that holds the actual contents. Fig. 6.4(a) shows a semi-full buffer, with the `tail` pointing to the next index to push to and `head` pointing to the next index to pop from. Pushing and popping happens in a circular fashion and only one producer thread and one consumer thread can call these respective routines.

Our context Fig. 6.4(b) shows the context in which these buffers are to be utilised. We have one thread, `T0` does the following actions in-order:

- reads new data from an input buffer, `in`,
- applies an arbitrary function, let's say `compute()`, to the read `data`,

- and, finally, writes its `result` to an output buffer, `out`.

We want to achieve this streaming pattern with two *properties*: 1) we can pipeline the code in `T0` and 2) our *II* must be independent of our `compute` function. The rest of this subsection is dedicated to discussing the process of materialising these properties. First, we show how a typical blocking implementation of `T0`, in §6.3.1.1. Then, we re-factor this implementation to produce a non-blocking and pipelinable implementation, in §6.3.1.2. Finally, we optimise this non-blocking implementation to achieve an implementation of `T0` whose *II* is independent of its computation, in §6.3.1.3.

Variable conventions For the rest of this subsection, we implement the following variable conventions for our code. We have two buffers (prefixed as `in` and `out`) with two atomic variables (suffixed as `Head` or `Tail`) and one non-atomic array (suffixed as `Array`).

6.3.1.1 Blocking implementation

Fig. 6.5 shows a typical implementation of `T0`. In T0, we have two inner while-loops: labelled as `consume` and `produce`. Additionally, the main while-loop (line 2) also has two Boolean variables, that are conditions of the two inner while-loops (line 3), and two integer variables `data`, which holds the data consumed from the `in` buffer (line 4), and `result`, which holds the data to push to `out` buffer after computation (line 15).

Each inner while-loop is a blocking implementation of the push and pop routines respectively. The `consume` loop attempts to pop `data` from the `in` buffer until it is successful and the `produce` loop attempts to push `result` to the `out` buffer until it is successful. Success is given by the fact that `in` buffer has an element to pop (`notEmpty` in line 9) or the `out` buffer has a space to push to (`notFull` in line 20). However, these inner while-loops suffer from one problem. Their termination depends on the runtime values of the current iteration. Hence, this implementation of `T0` leaves no room for loop-pipelining. To overcome these issues, we propose another implementation of `T0`.

6.3.1.2 Naive non-blocking implementation

Fig. 6.6 shows a new implementation of `T0`, which overcomes the issue of the previous version. Instead of a while-loop, we implement a for-loop (line 3) with a fixed number of iterations (represented as `ITER`). In the loop body, we attempt one pop from the `in` buffer and one push from the `out` buffer. We cannot execute these routines in strict order, because we may end up losing data if the pop succeeds but the push fails. So, we perform the `notEmpty` and `notFull` checks of the `in` and `out` buffer first (lines 4 to 7). If both conditions are satisfied, then we can proceed with the rest of the pop and push routines. Another way to visualise this implementation is that we are interleaving the

```
      atomic_int inHead = 0, inTail = 0; int inArray[SIZE];
     atomic_int outHead = 0, outTail = 0; int outArray[SIZE];
 1 void T0(){
 2  while(true){
 3   bool notEmpty = notFull = false;
 4   int data = 0;
 5   consume:  while(!notEmpty){
 6    int cTail = atomic_load_explicit(&inTail, memory_order_acquire);
 7    int cHead = atomic_load_explicit(&inHead, memory_order_relaxed);
 8    int nextHead = (cHead+1) % SIZE;
 9    notEmpty = cTail != cHead;
10    if(notEmpty) {
11     data = inArray[cHead];
12     atomic_store_explicit(&inHead, nextHead, memory_order_release);
13    }
14   }
15   int result = compute(data);
16   produce:  while(!notFull){
17    int pHead = atomic_load_explicit(&outHead, memory_order_acquire);
18    int pTail = atomic_load_explicit(&outTail, memory_order_relaxed);
19    int nextTail = (pTail+1) % SIZE;
20    bool notFull = pHead != nextTail;
21    if(notFull) {
22     outArray[pTail] = result;
23     atomic_store_explicit(&outTail, nextTail, memory_order_release);
24    }
25   }
26  }
27 }
```

Figure 6.5: A blocking implementation of `T0` in Fig. 6.4(b).

memory accesses of the pop and push routines of the `in` and `out` buffers in a way that does not risk losing packets.

This transformation allows the `non_blocking` loop to be pipelined, since its termination conditions does not dependent on the loop body. This loop has a fixed number of iterations and LegUp identifies `i` as an induction variable for loop-pipelining. Although this loop is pipelinable, it still suffers from a reduced version of the same problem. The `notFull` check of the `i+1`-th iteration (line 11) is dependent on the `i`-th iteration's release store of `outHead` (line 17). Since the `notFull` check directly impacts the `compute` function via a control dependency (line 12), in practice, only two consecutive iterations can overlap during runtime. Hence, its *II* is typically half of the `compute` function, which is a trend we highlighted in Chapter 5 for some memory access patterns. To overcome this issue, we must re-devise the memory access patterns of `T0` carefully.

```
        atomic_int inHead = inTail = 0; int inArray[SIZE];
      atomic_int outHead = outTail = 0; int outArray[SIZE];
```

```
void thread(){
 while(true){
  non_blocking:  for(int i=0; i < ITER; i++){
   int cTail = atomic_load_explicit(&inTail, memory_order_acquire);
   int cHead = atomic_load_explicit(&inHead, memory_order_relaxed);
   int pHead = atomic_load_explicit(&outHead, memory_order_acquire);
   int pTail = atomic_load_explicit(&outTail, memory_order_relaxed);
   int nextHead = (cHead+1) % SIZE;
   int nextTail = (pTail+1) % SIZE;
   bool notEmpty = cTail != cHead;
   bool notFull = pHead != nextTail;
   if(notEmpty && notFull) {
    int data = inArray[cTail];
    int result = compute(data);
    atomic_store_explicit(&inHead, nextHead, memory_order_release);
    outArray[pTail] = result;
    atomic_store_explicit(&outTail, nextTail, memory_order_release);
   }
  }
 }
}
```

Figure 6.6: A non-blocking stream implementation of T0 in Fig. 6.4(b) to enable loop-pipelining.

6.3.1.3 Optimised non-blocking implementation

Fig. 6.7 shows our final implementation of T0, which has several critical changes compared to Fig. 6.6. The key bottleneck of the previous version was that the each iteration required the most up-to-date information about whether out buffer is notFull. A simple work around to this problem is to move the notFull check's memory accesses out of the loop, as in lines 3 and 4. However, this change is insufficient since we could be writing to an out buffer that is already full and end up losing packets. To avoid over-optimising and inducing this bug into our implementation, we introduce a harness for this scenario by checking whether the out buffer has sufficient space before executing the loop. We do so with a function from the SPSC buffer's API, where details can be obtained in the Boost documentation [115]. We ensure that the out buffer has as many as spaces as iterations as our loop before executing the loop, as seen in lines 6 to 8.

With these changes, the II of our streaming_loop is not dependent on the latency of compute and hence we can overlap several consecutive iterations, instead of just two iterations. This implementation of T0 can achieve an II of three for any compute function that does not itself have inter-iteration aliasing memory dependencies. Moving forward,

```
      atomic_int inHead = 0, inTail = 0; int inArray[SIZE];
     atomic_int outHead = 0, outTail = 0; int outArray[SIZE];
void thread(){
 while(true){
  int pHead = atomic_load_explicit(&outHead, memory_order_acquire);
  int pTail = atomic_load_explicit(&outTail, memory_order_relaxed);
  int nextTail = (pTail+1) % SIZE;
  int spaceInBuffer = writeAvailable(pHead, pTail);
  if(spaceInBuffer >= ITER) {
   streaming_loop:  for(int i=0; i < ITER; i++){
    int cTail = atomic_load_explicit(&inTail, memory_order_acquire);
    int cHead = atomic_load_explicit(&inHead, memory_order_relaxed);
    int nextHead = (cHead+1) % SIZE;
    bool notEmpty = cTail != cHead;
    if(notEmpty) {
     int data = inArray[cTail];
     atomic_store_explicit(&inHead, nextHead, memory_order_release);
     outArray[pTail] = compute(data);
     atomic_store_explicit(&outTail, nextTail, memory_order_release);
    }
   }
  }
 }
}
```

Figure 6.7: A non-blocking stream implementation of `T0` in Fig. 6.4(b) with a pre-condition check to improve loop-pipelining capabilities.

this code template can be utilised to implement lock-free streaming on any application that can be partitioned into a streaming fashion, *i.e.* this coding template is generic for any streaming application.

6.3.2 Partitioning PageRank into a streaming pipeline

Now that we have a general coding template of lock-free buffers that can achieve one-to-one streaming with a low II, we can utilise this template to improve the runtime performance of PageRank. We transform the baseline PageRank, in §6.2.3, into a streaming pipeline based on the coding template of the previous section. We partition the baseline PageRank into pipelinable hardware stages connected via SPSC buffers. This way we can produce a lock-free streaming implementation of PageRank.

In Fig. 6.3, we discussed the three hardware stages generated by LegUp HLS from the C implementation of PageRank. Of this three stages, the first and last stages are non-trivial to pipeline. The first stage, `Fetch`, performs the memory accesses of the CSR matrices, which are data-dependent and hence difficult to pipeline. The last stage, `Spinning FAdd`

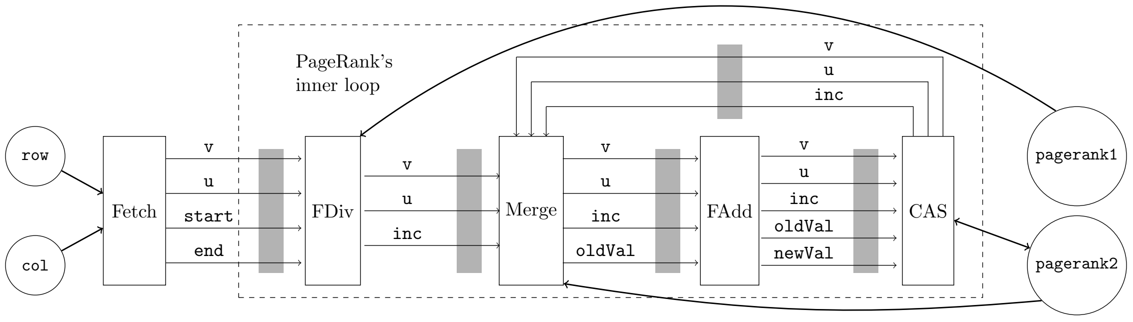


Figure 6.8: A streaming implementation of PageRank, where transparent rectangles with labels are pthreads (which also corresponds to a hardware module), grey-shaded rectangles are SPSC buffers and circles are memory elements.

`with CAS`, consists of the floating-point addition followed by a CAS operation, which implements the atomic floating-point addition. This loop is also difficult to pipeline because its termination condition depends on the CAS operation.

To achieve lock-free streaming, we must consider pipelining these two stages. The first stage only consists of four memory operations. Hence, the latency of this stage is small and comparable to the *II* we want to achieve. Consequently, we can avoid pipelining this stage since it is unlikely to affect performance. The last stage is too complex to pipeline and requires further partitioning into smaller stages. We split this stage into three stages:

- the `FAdd` stage, where we perform the floating-point addition;
- the `CAS` stage, where we perform the CAS operation;
- and, finally, the `Merge` stage, where we merge two buffer streams: 1) the stream from the `FDiv` stage with new packets and 2) the stream from the `CAS` stage with packets that failed their CAS.

Fig. 6.8 shows the streaming pipeline of PageRank that consists of five hardware stages. The first two stages are existing stages from the baseline implementation and the next three stages are the partitioned stages of the atomic floating-point addition. The grey-shaded rectangles are the SPSC buffers and each stage uses the same coding template as in Fig. 6.7, only with different `compute` functions. There is one key difference between the baseline and streaming implementations. The baseline only consists of one pthread and LegUp infers the three hardware stages automatically. In contrast, we must explicitly partition our streaming implementations into five pthreads each to achieve streaming.

Packets can fail their CAS operation at the `CAS` stage. In this case, these packets need to be fed back into the stream for retries. The `Merge` stage performs this task. At this stage, we must consider an important scenario where packets can arrive from both the `FDiv` stage and the feedback of the `CAS` stage simultaneously. To avoid deadlocks, we

Table 6.1: Design points for evaluating our lock-free streaming PageRank implementations.

Short name	Analysis	Streaming	Pipelining	MCM	Ref.	Impl.
Baseline	Global	No	No	*ppo*	§3.5.3	Fig. 6.3
LocalStreaming	Local	Yes	No	*mem-weak*	§3.5.3	Fig. 6.8
GlobalStreaming	Global	Yes	No	*ppo*	§5.4.2	Fig. 6.8
LocalStreamingPipe	Local	Yes	Yes	*mem-weak-pipe*	§5.4.1.2	Fig. 6.8
GlobalStreamingPipe	Global	Yes	Yes	*ppo-intra*, *ppo-inter*	§5.4.2	Fig. 6.8

must prioritise the feedback packets. If we do not process feedback packets immediately, both the feedback buffer and the output buffer of the `Merge` stage can become full at the same time. Once this happens, the stream will deadlock and never recover. We also do not pipeline the `Merge` stage since it only consists of memory operations and therefore its latency is small. In summary, we can achieve a lock-free streaming implementation of PageRank by partitioning the baseline into five stages (also five pthreads) and using SPSC buffers via our coding template.

6.3.3 Evaluating our streaming pipeline of PageRank

In this section, we evaluate our streaming implementation of PageRank against the baseline implementation of PageRank given discussed in 6.2.3.

6.3.3.1 Design points

We implement five different design points, as tabulated in Table 6.1. *Baseline* is our the Pannotia implementation of PageRank provided to LegUp, which discussed in Fig. 6.3. Since this baseline implementation also consists of atomic operations, we analyse this implementation via our global analysis. Then, we have *LocalStreaming*, *GlobalStreaming*, *LocalStreamingPipe* and *GlobalStreamingPipe* that implement the streaming PageRank implementation discussed in Fig. 6.8. The difference between these four design points is the type of analyses we apply and whether we enable loop pipelining.

6.3.3.2 Experimental setup

Our experimental setup is as follows:

- We only focus on evaluating PageRank based on weak atomics, since, SC atomics is generally slower than weak atomics;
- We set the SPSC buffer size (`SIZE`) as 128 and the number of iterations of the for-loop (`ITER`) as 32 for our streaming design points;
- We evaluate a partitioned DBLP co-author graph, where we pick integer values of nodes 2^N for $3 \leq N \leq 13$. Each N corresponds to a number of edges E and E

Table 6.2: Schedule latency (L) and initiation interval (II) of the different hardware stages (depicted in Fig. 6.8) for all design points in Table 6.1.

Stages	`Fetch`		`FDiv`		`Merge`		`FAdd`		`CAS`	
Metric	L	*II*	L	*II*	L	*II*	L	*II*	L	*II*
LocalStreaming	16	-	59	-	25	-	32	-	24	-
GlobalStreaming	16	-	58	-	23	-	32	-	23	-
LocalStreamingPipe	16	-	48	3	25	-	18	3	18	9
GlobalStreamingPipe	16	-	48	3	23	-	18	3	16	8

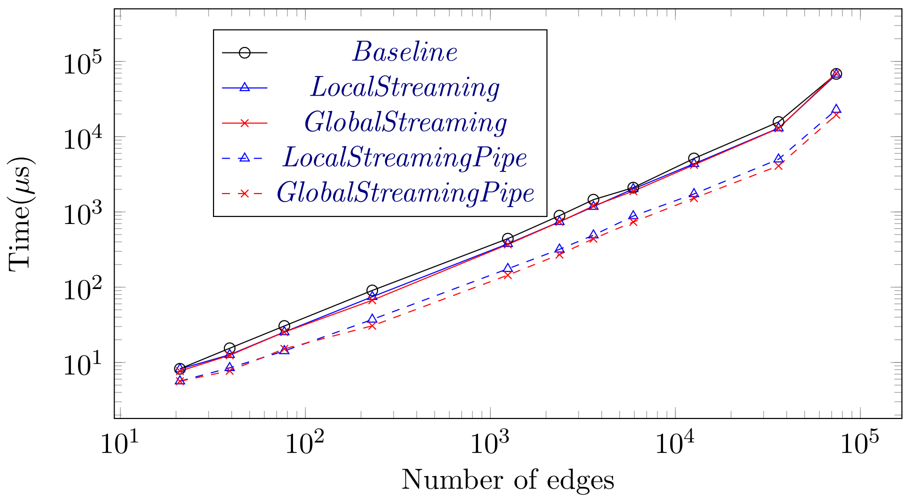


Figure 6.9: Hardware performance of our various design points against the number of edges, as we scale the number of nodes in the co-author graph.

increases monotonically as N increases. E dictates the size of the `col` array, which is the largest array and dominates the block RAM usage.

6.3.3.3 Runtime Performance

Latencies and initiation intervals affect performance Tab. 6.2 shows the latencies and initiation intervals of our four streaming design points. In comparison, the *Baseline* has a latency of *72 cycles*. There are three points to highlight from this table. Firstly, *LocalStreaming* and *GlobalStreaming* have higher total latencies across the five hardware stages (threads). This shows that our streaming template incur cycle overheads in terms of total latency. Secondly, the difference in latencies between *LocalStreaming* and *GlobalStreaming* are minimal, which shows that global analysis does not have a significant effect on latency. Thirdly, our loop pipelining extensions of thread-local and global analysis achieves good *II*s. *LocalStreamingPipe* and *GlobalStreamingPipe* are able to pipeline the `FDiv`, `FAdd` and `CAS` stages. The *II*s of `FDiv` and `FAdd` are the best achievable (three cycles), whereas the *II* of `CAS` is dependent on `compute` function (eight/nine cycles).

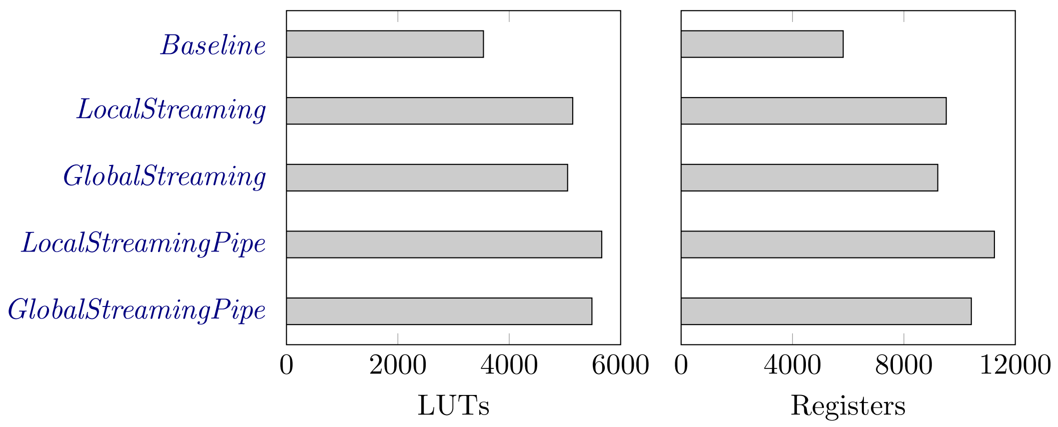


Figure 6.10: Average LUT and register usage for our design points in Tab. 6.1.

Overall runtimes Fig. 6.9 is a log-log plot that shows the runtime performance of our five design points against the number of edges (E). All our design points achieve a performance that is linear to the number of edges. However, there are significant offsets between these linear lines that require discussion.

Comparing our streaming implementations to baseline On average, the performance of *LocalStreaming* and *GlobalStreaming* is 1.2× faster than *Baseline*. Even though the total latencies of *LocalStreaming* and *GlobalStreaming* are larger than *Baseline*, the effects of partitioning and streaming PageRank boosts performance. In rare cases, these streaming pipelines are slower than *Baseline*. *GlobalStreaming* can be 2% slower than *Baseline*, but this is only due to clock frequency variations.

Enabling loop pipelining of our streaming implementations On average, *LocalStreamingPipe* and *GlobalStreamingPipe* is 2.5× and 3× faster than *Baseline*. Loop pipelining enables the stages with large latencies to overlap their iterations, as we see in *II* column of Table 6.2. We also see that, global analysis extracts further memory parallelism via loop pipelining. Hence, on average, *GlobalStreamingPipe* is 1.2× faster than *LocalStreamingPipe*. The true bottleneck of PageRank now becomes the `CAS` stage, rather than the `FDiv` and `FAdd` stages that are easily pipelinable.

6.3.3.4 Resource utilisation

LUT and register usage The LUT and register usage of our different design points do not change with the increase in number of edges (E). Fig. 6.10 shows the average LUT and register usage of all our design points. *Baseline* requires the least amount of LUTs and registers to implement. This is because of all streaming design points require additional hardware since the hardware stages are partitioned across five threads and these stages must also interfacing with SPSC buffers. For instance, *LocalStreamingPipe* requires 1.6×

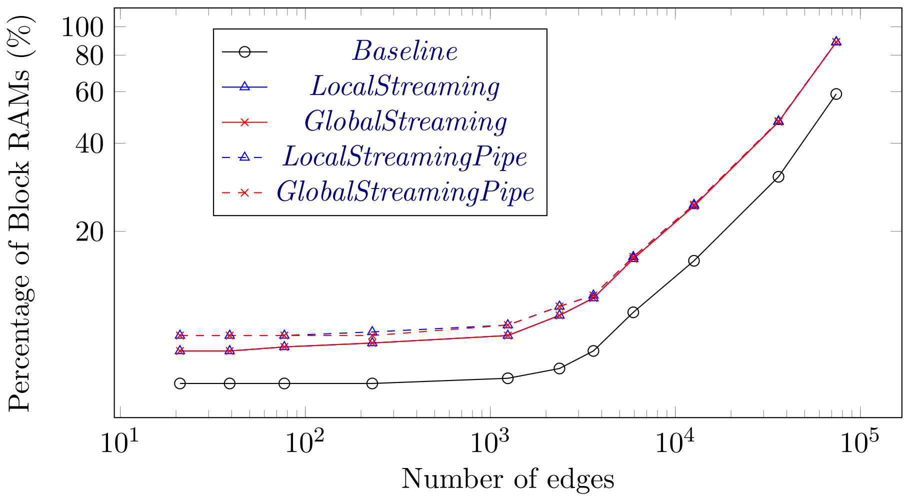


Figure 6.11: Block RAM utilisation of our various design points against the number of edges, as we scale the number of nodes in the co-author graph.

more LUTs and 1.9× more registers compared to *Baseline*. Additionally, the LUT and register usage of global analysis is marginally smaller than thread-local analysis, since its latencies and IIs are smaller than thread-local analysis.

Block RAM utilisation Fig. 6.11 shows the block RAM utilisation of all our design points, as we scale the number of edges. Firstly, we see that as the number of edges increases, the block RAM utilisation also increases. The shape of curve itself is uninteresting since it is data-dependent and much smaller than the theoretical upper bound of $|N| \leq |E|^2$. Secondly, the gap between the streaming design points and baseline is not constant. We would expect it to be constant to account for RAM usage of the buffers. However, this is not true because, for larger Es, Quartus decides to replicate the `col` array of our streaming design points.

6.3.4 Parallelising our implementations

Our each data point in the previous section is a standalone unit capable of executing any PageRank input graph, whether it is our baseline implementation with one pthread or our streaming implementations with five pthreads and five SPSC buffers. We refer to a collection of hardware capable of executing PageRank independently as a *compute unit*. Thus far, we only evaluate PageRank on a single compute unit. In this subsection, we replicate the number of compute units for all our implementations. To replicate compute units, we simply need to replicate the hardware pipelines and share the input graph and ranks with all compute units. There are several considerations during this replication process:

- For streaming implementations, we replicate all software threads and SPSC buffers.

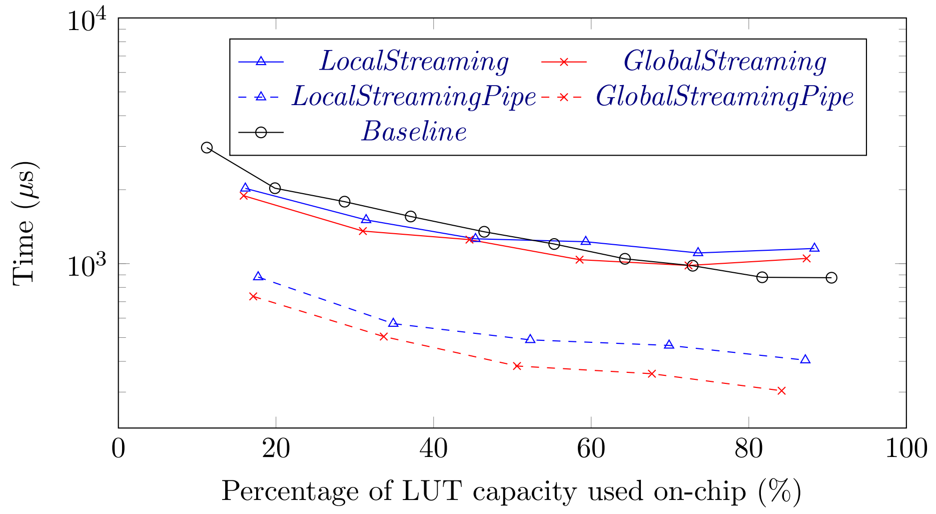


Figure 6.12: Runtime performance of our design points in Table 6.1 as we scale the number of compute units (P) for each design point.

- Each node is an individual task and hence we block-partition the nodes evenly across compute units and so we configure the `Fetch` stage of each compute unit appropriately.
- The CSR matrices and input ranks are shared among all compute units.
- We replicate the output ranks to ensure the CAS operations not to compete across compute units. Since each compute unit's output rank only holds a partial sum, we implement a highly-pipelined reduction after all compute units are joined. The time taken for this reduction is very minimal, compared to the overall runtime.

6.3.5 Evaluating our parallel Pagerank implementations

Now, we evaluate our parallel implementations of all five design points, discussed in Tab. 6.1. Our goal is to instantiate as many compute units as possible until we fill up the FPGA chip. We evaluate them on the same co-author graph as the previous section. We utilise the sub-graph with 1024 nodes and 5924 edges. We pick 1024 nodes instead of 8192 nodes, which is our maximum RAM capacity, to reduce the simulation time. As we scale the number of compute units, P, the LUT capacity of our FPGA chip maximises the fastest. Hence, Fig. 6.12 shows the runtime performance of our various design points versus the FPGA LUT capacity. The left-most data point of each design point is carried forward from §6.3.3, in which $P = 1$.

Scaling factor for design points are different The maximum P achievable vary according the design point. Since *Baseline*'s compute unit uses the least amount of LUTs, we can replicate it by 10 times. In contrast, we can only replicate *LocalStreaming* and

GlobalStreaming by 6 times, since these design points require additional hardware to implement hardware stages and buffer interfaces. Furthermore, loop pipelining incur more LUTs. Hence, we can only replicate *LocalStreamingPipe* and *GlobalStreamingPipe* by 5 times.

The runtimes of non-pipelined streaming saturates In terms of runtime performance, *LocalStreaming* and *GlobalStreaming* closely follow the performance of *Baseline*. Until $P = 3$, *LocalStreaming* is up to 1.4× faster than *Baseline*. Until $P = 4$, *GlobalStreaming* is up to 1.5× faster than *Baseline*. After these points, *Baseline* performs better since it becomes part of the Pareto-optimal curve. There are two reasons for this pattern. Firstly, since *Baseline* is smaller in LUT usage, we are able to replicate more compute units and eventually achieve better cycle counts than *LocalStreaming* and *GlobalStreaming*. Secondly, for similar Ps, *LocalStreaming* and *GlobalStreaming* cycle counts are better than *Baseline*, but these implementations suffer from higher clock frequency penalties.

The runtimes of pipelined streaming are high-performant When we enable loop-pipelining, our runtime performance is much better than *Baseline*. For the same number of compute units (P), *LocalStreamingPipe* and *GlobalStreamingPipe* is up to 4.4× and 5.4× faster than *Baseline*. Also, even though we are able to replicate more compute units for *Baseline*, its performance is not as good as our loop-pipelined streaming implementations. Hence, at maximum LUT capacity, *LocalStreamingPipe* is 2.2× than *Baseline*. Furthermore, global analysis provide additional speedups when utilised with loop pipelining. On average, *GlobalStreamingPipe* is 1.3× better than *LocalStreamingPipe*.

Summary In summary, at maximum LUT capacity, our best streaming implementation is at least 3× faster than any baseline implementation. It is also worth noting that our global analysis can cope with large real-world programs. At P=5, *GlobalStreamingPipe* has 25 pthreads (of which 15 threads require loop-pipelining), 25 lock-free buffers and 5 intermediate output ranks that are atomic arrays. Our analysis can generate the necessary ordering constraints of the entire program within 9 seconds.

6.4 Dynamic load-balancing via work-stealing

6.4.1 Introduction

The runtime performance of some design points in the previous subsection tend to saturate, as we scale the number of compute units. This suggests that the workload across the different threads are imbalanced and that our performance is dictated by the slowest thread. We investigate this hypothesis by characterising the input graph. All threads are given an equal number of nodes to process, but each node visits a variable number of

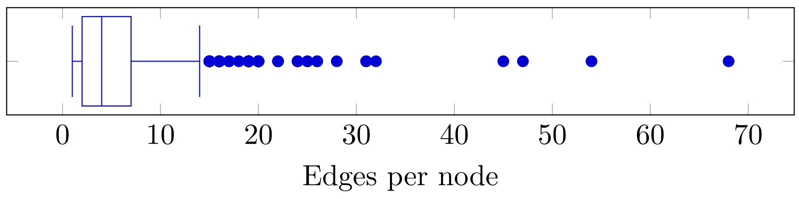


Figure 6.13: The distribution of edges to visit for each node of our input co-author graph, where the box shows the median and first and third quartiles, the whiskers extend the first and third quartiles by 1.5× of the inter-quartile range and dots are outliers.

edges depending to the input graph. Fig. 6.13 shows the edge distribution for all nodes of our input graph. Its median is four edges and its first and third quartiles have different ranges from the median (two and seven edges respectively). These statistics establish that our parallel implementations suffer from workload disparity. In fact, this disparity is more severe than expected because the upper whisker is 14 edges per node and also we have many outliers that are up to 68 edges. So it is hard to statically-partition an equal number of edges to process per thread.

One approach to reduce workload disparity is via *work-stealing* [141]. Work-stealing can perform dynamic load-balancing of irregular workloads, such as PageRank. In order for work-stealing to impact performance, the application must have the following properties:

- we must be able to define a task for each computation;
- and, either the execution time of a task is dynamic or a task can dynamically spawn new tasks, or both.

PageRank satisfies these properties because each node in the graph is a task and each task's execution time varies is dependent on the number of edges per task.

One of the first implementation of work-stealing on GPUs were by Cederman and Tsigas [142]. They use lock-free double-ended queues (*deques*) by Arora *et al.* [143], via OpenCL atomics, to load-balance a four-in-a-row game and an octree partitioner. We take inspiration from their work to implement work-stealing on FPGAs via HLS. We use a weakly consistent lock-free deque by Lê *et al.* [144] to improve the performance of our parallel PageRank implementations. In this subsection, we discuss the following:

- First, in §6.4.2, we discuss the functionality of a deque, how we optimise Lê *et al.*'s implementation to suit our purposes and how we implement work-stealing on PageRank using these deques.
- Next, in §6.4.4, we evaluate our work-stealing implementations of PageRank against our static-partitioned implementations.
- Finally, in §6.4.5, we also discuss some related works comparing this case study to previous work on work-stealing on FPGAs.

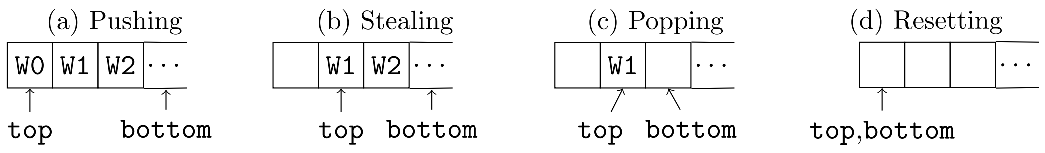


Figure 6.14: A sample execution of a deque.

6.4.2 Optimising the work-stealing deque for PageRank

6.4.2.1 Understanding the functionality of a deque

A double-ended qeuue (*deque*) consists of a non-atomic array protected by two atomic pointers that point to the `top` and `bottom` of the queue. A deque has three routines: `push`, `pop` and `steal`. The `push` and `pop` routines update the `bottom` pointer, whereas the `steal` routine updates `top` pointer. Each deque must be owned by one thread. Only the owner thread can push and pop tasks, which means it has exclusive write access to the `bottom` pointer. All other threads can `steal` tasks via the `top` pointer.

Fig. 6.14 is a sample execution of a deque. First, the owner thread pushes three tasks (`W0`, `W1` and `W2`) to the deque via the `bottom` pointer, as in Fig. 6.14(a). Then, a different thread steals a task (`W0`) from this deque via the `top` pointer, as in Fig. 6.14(b). Following this steal, the owner thread pops a task (`W2`) from the deque, updating the `bottom` pointer, as in Fig. 6.14(c). Finally, we are left with one task (`W1`) that can either be popped by the owner thread or stolen by any other thread. This corner case requires a CAS operation to achieve consensus, since both routines may be executed simultaneously by different threads. Eventually, the deque is emptied, as in Fig. 6.14(d).

6.4.2.2 Special properties of PageRank

We adapt the weakly consistent deque proposed by Lê *et al.* [144] for this case study. Our starting point is their original push, pop and steal routines, as in Fig. 1 of [144]. The full details of their implementation can also be found in their paper.

PageRank has two properties that allow further optimisation of this deque:

- PageRank's task is the node itself. The node index is sufficient to execute a task, which means we can simply return the pointer values and eliminate the non-atomic array all together. Eliminating the non-atomic array also means that we remove all memory fences in the original routines.
- PageRank does not spawn new tasks, unlike applications such as tree-traversal. The amount of tasks is fixed at compile time but each task's execution time is dynamic. Therefore, we can eliminate the `push` routine and simply pre-load these tasks.

```
int pop(atomic_int *top, atomic_int *bottom){
 int b = atomic_load(bottom, memory_order_relaxed) - 1; ❶
 atomic_store(bottom, b, memory_order_release); ❷
 int t = atomic_load(top,memory_order_acquire); ❸
 int x = EMPTY;
 if(t < b){
  x = b;
 } else {
  atomic_store(bottom, b+1, memory_order_release); ❹
 }
return x;
}
```

(a) The pop routine.

```
int steal(atomic_int *top, atomic_int *bottom){
 int t = atomic_load(top, memory_order_acquire); ❺
 int b = atomic_load(bottom, memory_order_acquire); ❻
 int x = EMPTY;
 if(t < b){
  x = t;
  if(!atomic_compare_exchange_strong_explicit( top, &t, t + 1,
     memory_order_release, memory_order_relaxed)); ❼
   x=ABORT;
 }
 return x;
}
```

(b) The steal routine.

Figure 6.15: Simplified deque routines of Lê *et al.*

6.4.2.3 Tailoring the deque routines

Based on these special properties of PageRank, Fig. 6.15 shows our optimised `pop` and `steal` routines. Only the owner thread can pop from a deque via updating `bottom` pointer, whereas all other threads attempting to steal from this deque compete with each other to update the `top` pointer. To guarantee this behaviour, the following release-acquire synchronisation pairs are put in place:

- all updates to the `bottom` pointer by the `pop` routine of the owner thread must be release stores, which are paired to the load acquire on the same location by the `steal` routines of any other thread, *i.e.* ❷ or ❹ synchronises with ❻;
- all updates to the `top` pointer by the `steal` routine of any thread must be a release store, which is paired with two other load acquires on the same location. Firstly, it is paired with a load acquire within the `pop` routine of the owner thread, *i.e.* ❼ can synchronises with ❺. Secondly, it paired with all load acquires within the `steal`

routine of any other thread attempting to steal from the same deque, *i.e.* ❼ can synchronises with ❺.x

The `pop` routine The pop routine, shown in Fig. 6.15(a), has the following sequence of events. First, the latest `bottom` value is read and reserved by the owner thread via a decrement and update, which is a release store. Then, the latest `top` value is read via a load acquire (line 1.4). Finally, if the latest `top` value is larger than the latest `bottom` value (line 1.6), then there is more one than one element in the deque and the latest `bottom` value is returned (line 1.7 and 1.11). Otherwise, the reservation of the pop content is reverted via a release store (line 1.9) and the deque is treated as if its empty (line 1.5 and 1.11). Note that we optimised away part of the original deque's pop routine. As discussed in our sample execution, in Fig. 6.14(c), the last element of a deque can either be popped or stolen. To ensure correct behaviour, a CAS operation is required within both these routines. However, we decided statically to allow the steal routine to always win this duel. Hence, we can remove the CAS operation within the pop routine.

The `steal` routine The steal routine, shown in Fig. 6.15(b), has the following sequence of events. First, the latest `top` and `bottom` values are read via load acquires (lines 2.2 and 2.3). Then, we check whether there are any elements in the deque (line 2.5). If so, then attempt to steal the the `top` pointer via a CAS operation, which is a release CAS (line 2.7). In the original deque routine, the CAS is attempted until it succeeds. Instead, we only attempt the CAS once and simply return the `ABORT` signal if it fails. We use this signal to perform other useful tasks within the thread, in the event of CAS failures.

6.4.3 Incorporating these deques into PageRank

In this previous subsection, we presented the functionality of a work-stealing deque and how we optimise an original version of Lê *et al.*'s deque to suit our PageRank implementation. Now, we discuss how we use this simplified deque in our concurrent program that implements PageRank. We discuss our work-stealing PageRank implementation with two compute units ($P = 2$), which generalises to any P.

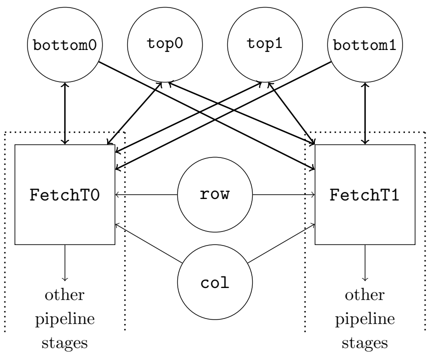
Fig. 6.16 shows how we implement our work-stealing PageRank implementation for $P = 2$. For all our implementations, we only require extending the first hardware stage, `Fetch`, to implement work-stealing. First, we instantiate a deque for each thread *i.e.* every thread owns a deque. The ownership pattern for the atomic pointers is shown by directional arrows in Fig. 6.16(b). Only `FetchT0` can write to `bottom0` and `FetchT1` can write to `bottom1` respectively. We also do not require the task arrays, since PageRank only requires the pointer values. For the same reason, we can pre-load each thread with tasks via simple initialisation of the respective `top` and `bottom` pointers.

```
atomic_int bottom0 = 0; atomic_int top0 = NODES/2;
atomic_int bottom1 = NODES/2; atomic_int top1 = NODES;
```

```
1.1 void FetchT0(){
1.2   while(true){
1.3     int i = pop(&top0, &bottom0);
1.4     if(i==EMPTY)
1.5       i = steal(&top1, &bottom1);
1.6     if(i==EMPTY||i==ABORT) break;
1.7     process(i);
    }
```

```
2.1 void FetchT1(){
2.2   while(true){
2.3     int i = pop(&top1, &bottom1);
2.4     if(i==EMPTY)
2.5       i = steal(&top0, &bottom0);
2.6     if(i==EMPTY||i==ABORT) break;
2.7     process(i);
    }
```

(a) Psuedo-code



(b) Generated hardware

Figure 6.16: Work-stealing PageRank implementation for two compute units ($P = 2$).

For our case study, we block-partition PageRank, as seen in with the initialisation in Fig. 6.16(a). Then, we set up each thread to first attempt a pop from its own deque (lines 1.3 and 2.3). If that fails, then the thread attempts a steal from the other deque (lines 1.5 and 2.5). For larger Ps, we can generalise this pattern, where each thread to attempt another steal to subsequent deques if the previous steal fails. This is the reason we do not keep attempting CASes to the same deque in the `steal` routine of Fig. 6.15(b). We simply decide to steal from another deque, since other deques may have more work to do and also we ensure that the program is achieving overall progress. Finally, if a task was acquired, then it is processed by the remaining hardware stages of the compute unit (lines 1.7 and 2.7). Otherwise, the loop terminates (lines 1.6 and 2.6).

A final note about the `Fetch` stage is that we chose not partition this stage for loop pipelining because it only consists of memory operations and had a small latency. This decision fits well with our work-stealing implementation, though for a different reason. Work-stealing introduces complex control structures within the loop and its loop termination is based on the dynamic memory values, which makes it very difficult to pipeline. This stage, however, provides packets to the rest of the pipeline that is highly pipelined.

Table 6.3: Extending the design points in Table 6.1 to support work-stealing.

Short name	Original implementation	MCM	Ref.
LocalStreamingWS	*LocalStreaming*	*mem-weak*	§3.5.3
GlobalStreamingWS	*GlobalStreaming*	*ppo*	§5.4.2
LocalStreamingPipeWS	*LocalStreamingPipe*	*mem-weak-pipe*	§5.4.1.2
GlobalStreamingPipeWS	*GlobalStreamingPipe*	*ppo-intra*, *ppo-inter*	§5.4.2

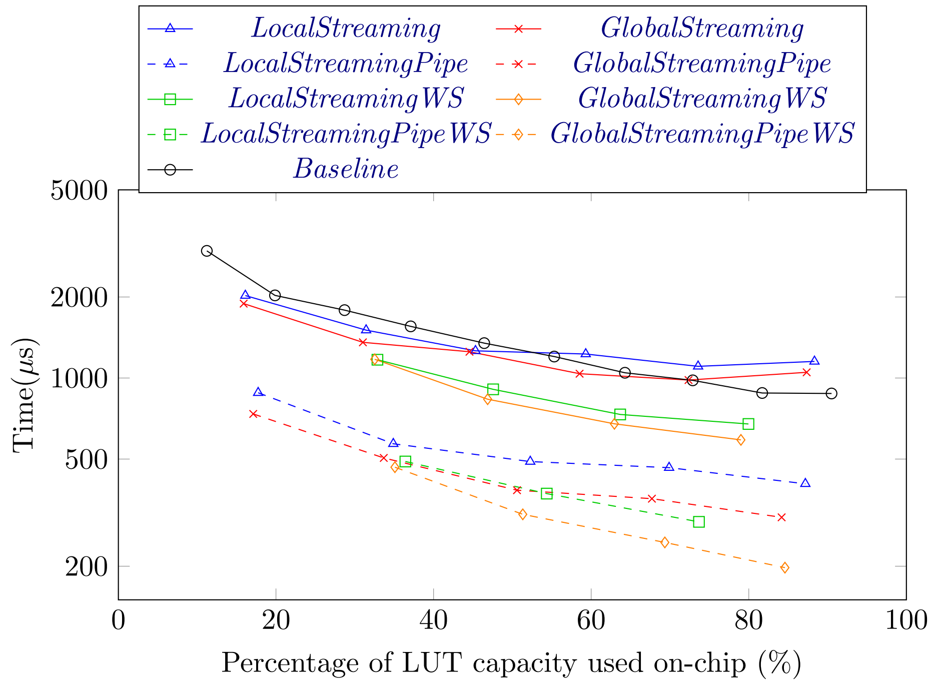


Figure 6.17: Execution time versus hardware utilisation of all design points given in Tab. 6.1 and 6.3, as we scale the number of parallel hardware units (P).

6.4.4 Evaluating work-stealing on PageRank

We evaluate our work-stealing implementations against the previous implementations of PageRank. Hence, we carry forward five design points from Table 6.1, which are statically-partitioned. Then, the four streaming design points are extended to support work-stealing, as shown in Table 6.3. Fig. 6.17 shows the performance of all our nine design points. Again, our goal is to maximise LUT capacity of all design points via scaling the number of compute units (P).

Work-stealing always improves performance Work-stealing allows better workload distribution, reducing workload disparity. Consequently, work-stealing improves the overall performance. *LocalStreamingWS* and *GlobalStreamingWS* are 1.5× faster than *LocalStreaming* and *GlobalStreaming* respectively. Also, at higher values of P, the performance of *LocalStreamingWS* and *GlobalStreamingWS* does not saturate or become worse than

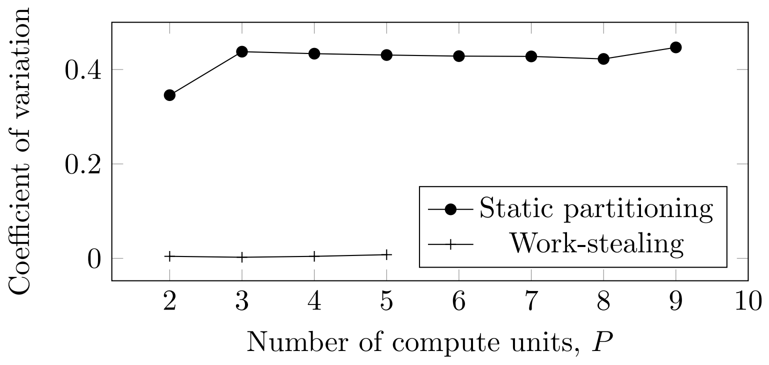


Figure 6.18: Workload distribution (forward links visited) across multiple compute units for the naïve and work-stealing cases, where $N = 1024$ and $E = 5924$.

Baseline, unlike *LocalStreaming* and *GlobalStreaming*. Furthermore, work-stealing also enhances the performance of our pipelined designs. Although the hardware stage in which we implement work-stealing is not loop-pipelined, workload distribution improves overall performance. On average, we see that *LocalStreamingPipeWS* and *GlobalStreamingPipeWS* is 1.7× and 1.6× faster than *LocalStreamingPipe* and *GlobalStreamingPipe*.

Area overheads are higher for work-stealing but worthwhile Work-stealing always incurs additional area overheads. Hence, we can fit fewer compute units as we scale P for work-stealing implementations. We can fit up to 6 compute units for *LocalStreaming* and *GlobalStreaming*. In contrast, we only can fit 5 compute units for *LocalStreamingWS* and *GlobalStreamingWS*. However, at maximum LUT capacity, both these design points are 1.7× faster than *LocalStreaming* and *GlobalStreaming* respectively. Additionally, the LUT utilisation of thread-local analysis is higher than global analysis. This is most evident because we cannot implement $P = 5$ for *LocalStreamingPipeWS*, but we can do so for *GlobalStreamingPipeWS*.

Workload distribution demonstrates the capabilities of work-stealing Fig. 6.18 shows the workload distribution of our statically-partitioned and work-stealing implementations. We plot the coefficient-of-variation (CoV), which is the standard deviation divided by the mean, versus the number of compute units (P). The larger the CoV, the larger the workload disparity is across threads. Work-stealing improves the CoV by two orders of magnitude.

Summary Overall, we show that incorporating work-stealing improves PageRank's performance since we are able to distribute work more evenly across threads. We were able to implement non-trivial memory synchronisation using atomics to share the PageRank workload across threads with relatively small area overheads. At maximum LUT capacity, *GlobalStreamingPipeWS* is the best-performing design point.

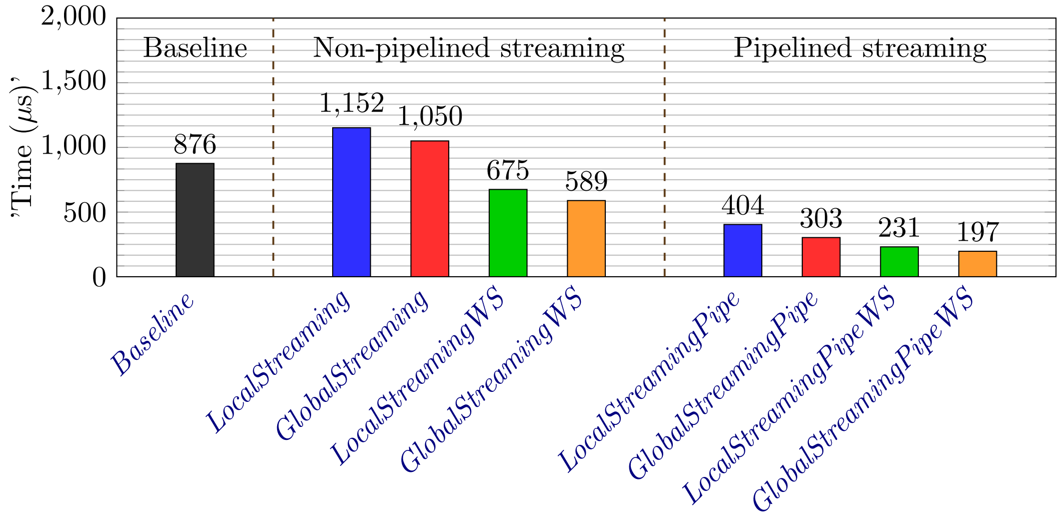


Figure 6.19: Runtime performance of PageRank for all design points in Table 6.3 at maximum area capacity.

6.4.5 Related work

Work-stealing has been previously implemented on FPGAs by us [145]. The difference between our case study in this thesis and our previous work is as follows:

- We implement work-stealing on the PageRank algorithm, instead of k-means clustering (KMC). KMC is different to PageRank since it is a tree traversal, instead of a graph application. From the point of view of work-stealing properties, PageRank does not spawn new tasks, unlike KMC. Furthermore, KMC requires tracking of various stacks and heaps via pointers. These pointers are part of its task description and therefore we cannot remove the task array, as we do with PageRank.
- We use a work-stealing double-ended queue that supports weak atomics. This is was not possible with our previous work, since Altera OpenCL does not support weak atomics. This thesis is the first to enable synthesis of weak atomics.
- Our PageRank application is optimised to support streaming and loop-pipelining whereas the KMC implementation was neither streamed nor loop-pipelined.

6.5 Conclusion

This chapter presents a case study to showcase the benefits of using fine-grained atomics to improve the performance of an HLS implementation of Google's PageRank algorithm. We implement lock-free streaming and dynamic load-balancing on PageRank and then apply our various analyses to further improve its performance. Fig. 6.19 recaps the performance of our different implementations of PageRank at maximum LUT capacity. We see that converting the baseline implementation of PageRank (*Baseline*) into a streaming implementation does not improve performance. *LocalStreaming* and *GlobalStreaming* is 30% and 20% slower than *Baseline*. However, their performances improve when we enable loop pipelining. *LocalStreamingPipe* and *GlobalStreamingPipe* is 2.2× and 2.9× faster than *Baseline*. Independently, PageRank performance also improves when we implement work-stealing since we can reduce workload disparity. *LocalStreamingWS* and *GlobalStreamingWS* is 1.3× and 1.5× faster than *Baseline*. Overall, we achieve the best performance when we enable both loop pipelining and work-stealing. *LocalStreamingPipeWS* and *GlobalStreamingPipeWS* is 3.8× and 4.4× faster than *Baseline*. It is also worth noting that, at $P = 5$, our global analysis can generate the necessary ordering constraints of *GlobalStreamingPipeWS*, which consists of 30 threads, within 12 seconds.

7. Conclusion

7.1 Summary and key contributions

In this thesis, we have explored the possibility of synthesising fine-grained and weakly consistent C concurrency via high-level synthesis (HLS). As the appeal of software concurrency increases, HLS tools are beginning to adopt concurrency to expand their user base and synthesisable code base. However, HLS tools still rely on the synthesis methods of sequential C programs to synthesise concurrent C programs. Although this approach makes the support for concurrent programs an incremental process, it can lead to sub-optimal hardware architectures. One instance of such a problem is that memory scheduling of each concurrent thread is executed independently based on the scheduling rules of a sequential program. Within a sequential program, only aliasing memory orderings must be preserved. Although this rule is sufficient for a sequential program, it is too weak to support any form of memory synchronisation across threads of a concurrent program, thereby introducing the need for locks. Locks ensure that shared memory resources are only accessed by one thread at time. Even though locks ensure correct memory behaviour, they tend to serialise shared memory accesses and are susceptible to deadlocks.

Instead of lock-based synchronisation, the C memory model describes the possibility of *lock-free* synchronisation via *fine-grained* atomics. Atomics, when used in a race-free manner, can support memory synchronisation across threads without locks, thereby avoiding their memory serialisation effects. Despite the benefits of atomics, HLS tools either do not support atomics at all or do so inefficiently whilst discouraging their use. The standard approach of implementing atomics via HLS is to wrap locks around atomic accesses [8]. This approach, again, ensures correctness but re-introduces the drawbacks of using locks. Instead, in this thesis, we proposed a HLS-friendly method of implementing atomics. Now, we discuss our contributions and achievements of this thesis on per-chapter basis and how they address our research questions discussed in page 21.

In **Chapter 3**, we demonstrated how current HLS memory models are too weak to support atomics. Then, we proposed treating the synthesis of atomics as a scheduling problem, where we identify atomic accesses and subject them to additional scheduling rules within a concurrent thread. These scheduling rules thereby generate additional intra-thread memory constraints that each atomic access must adhere to during memory scheduling. By doing so, we addressed **RQ 1** whereby we devise a method that does not involve wrapping locks around memory accesses to implement atomics. Our method achieved an average speedup of 7.5×, compared to the state-of-the-art HLS method of synthesising atomics, on our set of experiments. Since we implement atomics via scheduling constraints, we can also tailor our scheduling rules to take into consideration the consistency mode of each atomic access. By doing so, our method is the first method capable of synthesising weak atomics via HLS directly onto hardware, which addresses **RQ 2**. Although weak atomics are capable of improving memory parallelism, their implementation can be complex and error-prone. Hence, we verified our implementation via automated model checking to ensure that our hardware execute atomics correctly. We achieved a further 1.6× speedup when our analysis is sensitive to weak atomics, for our set of experiments.

In **Chapter 4**, we first demonstrated how the current approach of generating memory constraints for a concurrent thread only based its own memory accesses, *i.e.* thread-local analysis, is conservative. Although thread-local analysis ensures that memory constraints are intra-threaded, we argue that memory scheduling based on global analysis can also generate intra-thread memory constraints and be more efficient. Hence, we proposed a global analysis that analyses all memory accesses of a concurrent program, including atomics, to generate the intra-thread memory constraints for memory scheduling of individual threads, which addresses **RQ 3**. We take advantage of the fact that the C memory model describes lock-free synchronisation globally to devise a method that enumerates all possible execution paths of a given concurrent program. Since the number of executions of a concurrent program can increase exponentially, depending on its memory accesses and thread counts, we also proposed an optimisation to improve our analysis times, making it more scalable and practical for reasonably-sized programs. We achieved an average speedup of 3.4× using global analysis, compared to thread-local analysis, on our set of experiments.

In **Chapter 5**, we explored the possibility of supporting atomics in the context of loop pipelining. Since we synthesise atomics via HLS scheduling constraints, we argued that supporting loop pipelining is possible since it is also implemented using scheduling constraints. The intra-thread constraints we generate to support atomics can be extended to also generate inter-iteration constraints required to support loop pipelining. Therefore, we formalised the current support of loop pipelining for HLS memory models and showed why they are too weak to support atomics. Accordingly, we proposed extensions to both our thread-local and global analyses, of Chapters 3 and 4, to support loop pipelining of program with atomics, which addresses **RQ 4**. We achieved an average speedup of 1.4× when enabling loop pipelining on our set of experiments. We also achieved an average speedup of 5.7× when enabling loop pipelining on a set of compute-dominant experiments, which consists of both memory and compute workloads, demonstrating that loop pipelining is better suited to programs with heavy computation.

Finally, in **Chapter 6**, we employed all our novel analyses of the previous chapters to tackle a real-world example to showcase the benefits of our work. Hence, we chose to optimise an HLS-based implementation of Google's PageRank algorithm, which is an important and well-known graph application. We optimise PageRank in two ways. Firstly, we partitioned the application into small and lean hardware stages and implemented lock-free streaming between these stages. We showed that a lock-free, streamed and pipelined implementation of PageRank is 2.9× faster than its HLS baseline implementation, at maximum chip capacity. Secondly, we implemented lock-free work-stealing to load-balance PageRank's workload evenly across the independent compute units. Since PageRank is an irregular workload, it is hard to statically partition its workload evenly. Therefore, we implemented a lock-free double-ended queue to enable work sharing via non-trivial but inexpensive atomic synchronisation across threads. We showed that work-stealing improves the performance of PageRank by 1.5×, at maximum chip capacity.

7.2 Limitations

Our work may suffers from several limitations.

Assumptions We assume that HLS tools compile all shared memory variables and arrays as on-chip direct-mapped memories, which limits us in three ways. Firstly, we are unable to explore the possibility that shared memories can be targeted to off-chip memory technologies, such as DRAMs. Secondly, we are unable to understand how atomics can be implemented if our on-chip memories co-exist and interact with hardened processor cores that are connected to the FPGA fabric via high-performant and customised communication buses. Thirdly, we are unable to explore memory subsystems with capabilities to synthesise on-chip caches and write buffers, since we assume all shared memories are directly mapped. In the future, these technologies may become part of HLS of concurrent programs thereby challenging our assumptions in many ways and requiring future investigations to handle these complexities.

LegUp implementation We implemented our analyses in LegUp, which has its own limitations. Since LegUp's pure hardware flow enforces that shared memory can only be targeted on-chip, we could only explore the implementation of atomics only via instruction reorderings, and could not gain insights on off-chip accesses where the indivisibility of atomic operations may not be guaranteed. In addition, LegUp also only executes one basic block at a time, which means it does not take into account of the inter-basic block reorderings that our analyses may discover, limiting the achievable performance on the hardware generated of our analyses.

Analyses and evaluation In terms of analyses, there are several observations that may be weaknesses of our work. Firstly, the ordering constraints generated by thread-local analysis, especially in the context of loop pipelining, may cause the underlying LP solver of LegUp to run out of memory. Although our analyses can generate the necessary constraints for these lock-free programs, in rare cases, the complexity and size of these programs may exceeds the scheduling capabilities of current HLS tools. Secondly, our global analysis still scales exponentially, in the worst case, as discussed in §4.7.4.1. The scaling of our global analysis is not only reliant on the input program size, but also its memory access pattern. Thirdly, since targeting lock-free data structures via HLS is relatively new, we were unable to leverage on any existing FPGA-based benchmarks to evaluate our work. Instead, we hand picked several data structures and tested their performance on several commonly-used data transfer patterns.

7.3 Future work

There are several interesting directions that arise from our work, which we present in the form of research questions:

- **Can we further improve memory scheduling of lock-free programs?** There are many possible ways in which memory parallelism can be further exploited. For example, we focus on memory scheduling based on the assumption that all basic blocks are executed in program sequence. However, during runtime, not all basic blocks will be executed and they can also take different control paths. Also, LegUp executes basic blocks one at a time but, in practice, several basic blocks may executed at the same time on different HLS tools. Furthermore, CAS operations have two consistency modes: one if the CAS succeeds and another if it fails. We only consider the succeeding case, but the failure case is generally weaker. Additionally, there is also a weak version of CAS operations, which we did not consider implementing. From these different perspectives, our scheduling constraints and runtime results could be improved.
- **Can we implement fine-grained C concurrency in the context of caches?** A key assumption of our work, which we established in the Introduction chapter, is that we assume that C shared memory constructs are synthesised directly to on-chip memories without caches or write buffers. As of now, this is the current standard practice of HLS tools that support concurrent C programs. However, it is an interesting and open question as to how to support fine-grained atomics in the

context of HLS tools that can generate architectures with caches and buffers, which would require ensuring both the atomicity and ordering properties of atomics in a efficient and high-performance manner.

- **Can we improve our correctness guarantees?** Since HLS compilers have access to the entire program at compile-time, unlike software compilers, it may be possible to check the correctness of our scheduling rules for particular program instances. Currently, our scheduling rules are general and this forces our model checking tool to find counter-examples amongst all programs for a bounded number of memory events. This formulation can have a large search space. Instead, could we provide Alloy with the specific memory access patterns of a program to reduce the state exponentiation?
- **Can we improve global analysis to tackle worst-case behaviour?** We propose an optimisation to improve the scalability of our global analysis in §4.4. Although this optimisation improves analysis times of realistic programs, our analysis still may not be able to cope with certain worst-case behaviours, such as the ones described in §4.7.4.1. It remains an open question as to whether we can improve our analysis times by eliminating all possible worst-case behaviours that leads to exponential path enumeration. In particular, there may be an interesting trade-off between analysis times and memory parallelism for worst-case behaviours.

7.4 Final remarks

In summary, our work on synthesising fine-grained and weakly consistent C atomics via HLS broadens the horizon of applications that can be targeted to reconfigurable architectures. Our work opens up the possibilities of synthesising generic lock-free programs, such as our case study whereby an HLS-based implementation of PageRank can be streamed, pipelined and load-balanced via the use of lock-free data structures. As software applications grow in complexity and the appeal for fine-grained concurrency increases, we hope that our work can be an avenue to compile the latest advances and applications of lock-free programming directly onto hardware.

Bibliography

[1] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, “An Introduction to High-Level Synthesis,” *IEEE Design and Test of Computers*, vol. 26, pp. 8–17, July 2009.

[2] INtel, *Intel FPGA SDK for OpenCL Pro Edition - Programming Guide*, 2019.

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

[4] IEEE, “POSIX Threads,” IEEE Std 1003.1c, 1995.

[5] Khronos Group, *The OpenCL Specification.* Version 2.0, 2015. Available at `https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf`.

[6] LegUp Computing Inc., “LegUp 5.1 Documentation..” `http://bit.ly/2D7VrQ0`, 2017.

[7] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[8] J. Choi, S. Brown, and J. Anderson, “From software threads to parallel hardware in high-level synthesis for FPGAs,” in *2013 International Conference on Field-Programmable Technology (FPT)*, pp. 270–277, Dec 2013.

[9] All experiment data is available at `https://doi.org/10.5281/zenodo.3146758`.

[10] K. Rupp, “42 Years of Microprocessor Trend Data.” Available at `https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`.

[11] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[12] S. Borkar and A. A. Chien, “The Future of Microprocessors,” *Commun. ACM*, vol. 54, pp. 67–77, May 2011.

[13] H. Sutter, “The free lunch is over: A fundamental turn toward concurrency in software,” 2005. Available at `http://www.gotw.ca/publications/concurrency-ddj.htm`.

[14] T. Rauber and G. Rünger, *Parallel programming: For multicore and cluster systems.* Springer Science & Business Media, 2013.

[15] G. E. Moore, "Cramming more components onto integrated circuits," *IEEE Solid-State Circuits Society Newsletter*, vol. 11, pp. 33–35, Sep. 2006.

[16] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, pp. 256–268, Oct 1974.

[17] R. Merritt, "ARM CTO: power surge could create dark silicon," 2009. Available at `https://www.eetimes.com/document.asp?doc_id=1172049#`.

[18] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs?," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 225–236, IEEE Computer Society, 2010.

[19] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation Cores: Reducing the Energy of Mature Computations," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 205–218, ACM, 2010.

[20] S. Yehia, S. Girbal, H. Berry, and O. Temam, "Reconciling specialization and flexibility through compound circuits," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pp. 277–288, Feb 2009.

[21] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, pp. 203–215, Feb 2007.

[22] S. M. S. Trimberger, "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology," *IEEE Solid-State Circuits Magazine*, vol. 10, pp. 16–29, Spring 2018.

[23] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating Compute-Intensive Applications with GPUs and FPGAs," in *2008 Symposium on Application Specific Processors*, pp. 101–107, June 2008.

[24] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Comput. Surv.*, vol. 34, pp. 171–210, June 2002.

[25] Intel Corporation, "Intel completes Acquisition of Altera," 2015. Available at `https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/`.

[26] Amazon EC, "F1 instances: Run custom FPGAs in the AWS cloud, 2017." Available at `https://aws.amazon.com/ec2/instance-types/f1`.

[27] A. Putnam *et al.*, "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services," pp. 13–24, 2014.

[28] P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit.* Springer Publishing Company, Incorporated, 1st ed., 2010.

[29] Berkeley Design Technology, Inc., "An Independent Evaluation of: The AutoESL AutoPilot High-Level Synthesis Tool," tech. rep.

[30] F. J. Pollack, "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies (Keynote Address)(Abstract Only)," in *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, (Washington, DC, USA), pp. 2–, IEEE Computer Society, 1999.

[31] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2011.

[32] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: a tutorial," *Computer*, vol. 29, pp. 66–76, Dec 1996.

[33] M. S. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," vol. 39, (New York, NY, USA), pp. 244–256, ACM, Apr. 2004.

[34] A. Williams, *C++ Concurrency in Action.* Manning, 2017.

[35] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming.* Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[36] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, pp. 1591–1604, Oct 2016.

[37] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems, vol. 30, pp. 473–491, April 2011.

[38] Xilinx, *Vivado Design Suite - HLx Edition.* Available at `https://www.xilinx.com/products/design-tools/vivado.html`.

[39] Calypto Design Systems, "Catapult: Product Family Overview," 2014. Available at `http://calypto.com/en/products/catapult/overview`.

[40] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing Modular Hardware Accelerators in C with ROCCC 2.0," in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 127–134, May 2010.

[41] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, (New York, NY, USA), pp. 33–36, ACM, 2011.

[42] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *2013 23rd International Conference on Field programmable Logic and Applications*, pp. 1–4, Sep. 2013.

[43] R. Nane, V. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels, "DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pp. 619–622, Aug 2012.

[44] S. Singh and D. J. Greaves, "Kiwi: Synthesis of FPGA Circuits from Parallel Programs," in *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pp. 3–12, April 2008.

[45] D. Pellerin and S. Thibault, *Practical FPGA Programming in C.* Upper Saddle River, NJ, USA: Prentice Hall Press, first ed., 2005.

[46] C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM*, vol. 21, pp. 666–677, Aug. 1978.

[47] Impulse Accelerated Technologies, *Impulse CoDeveloper C-to-FPGA Tools.* Available at `http://www.impulseaccelerated.com/productsuniversal.htm`.

[48] Maxeler Technologies, "MaxCompiler." Available at `https://www.maxeler.com/products/software/maxcompiler/`.

[49] G. Arnout, "SystemC Standard," in *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, ASP-DAC '00, (New York, NY, USA), pp. 573–578, ACM, 2000.

[50] D. C. Black and J. Donovan, *SystemC: From the Ground Up.* Berlin, Heidelberg: Springer-Verlag, 2005.

[51] M. Meredith, "High-level SystemC synthesis with Forte's Cynthesizer," in *High-Level Synthesis*, pp. 75–97, Dordrecht: Springer, 2008. Available at `http://www.forteds.com/products/cynthesizer.asp`.

[52] Cadence, "C-to-Silicon Compiler." Available at `http://www.cadence.com/products/sd/siliconcompiler/pages/default.aspx`.

[53] F. Ghenassia, *Transaction-Level Modeling with Systemc: Tlm Concepts and Applications for Embedded Systems.* Berlin, Heidelberg: Springer-Verlag, 2006.

[54] M. Aubury, I. Page, G. Randall, J. Saul, and R. Watts, *Handel-C language reference guide*, 1996.

[55] Mentor Graphics, "DK Design Suite: Handel-C to FPGA for Algorithm Design," 2010. Available at `http://www.mentor.com/products/fpga/handel-c/dk-design-suite`.

[56] "Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications," in *Proceedings of the Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '04, (Washington, DC, USA), pp. 69–70, IEEE Computer Society, 2004.

[57] R. S. Nikhil, "Bluespec: A general-purpose approach to high-level synthesis based on parallel atomic transactions," in *High-Level Synthesis*, pp. 129–146, Dordrecht: Springer, 2008.

[58] R. S. Nikhil and Arvind, "What is Bluespec?," *SIGDA Newsl.*, vol. 39, Jan. 2009.

[59] D. L. Rosenband and Arvind, "Hardware Synthesis from Guarded Atomic Actions with Performance Specifications," in *Proceedings of the 2005 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '05, (Washington, DC, USA), pp. 784–791, IEEE Computer Society, 2005.

[60] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the Scala programming language," tech. rep., EPFL Lausanne, Switzerland, 2004.

[61] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Aviienis, J. Wawrzynek, and K. Asanovi, “Chisel: Constructing hardware in a Scala embedded language,” in *DAC Design Automation Conference 2012*, pp. 1212–1221, June 2012.

[62] Xilinx, *AXI Reference Guide, UG761 (v13. 1)*, 2011. Available at `https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug761_axi_reference_guide.pdf`.

[63] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st ed., 1994.

[64] E. S. Chung, J. C. Hoe, and K. Mai, “Coram: An in-fabric memory architecture for fpga-based computing,” in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA ’11, (New York, NY, USA), pp. 97–106, ACM, 2011.

[65] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer, “Leap Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic,” in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA ’11, (New York, NY, USA), pp. 25–28, ACM, 2011.

[66] F. Winterstein, K. Fleming, H.-J. Yang, S. Bayliss, and G. Constantinides, “MATCHUP: Memory Abstractions for Heap Manipulating Programs,” in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA ’15, (New York, NY, USA), pp. 136–145, ACM, 2015.

[67] Intel, *Intel FPGA SDK for OpenCL Pro Edition - Best Practices Guide*, 2019.

[68] 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 *2015 International Conference on Field Programmable Technology (FPT)*, pp. 152–159, Dec 2015.

[69] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO ’04, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.

[70] LLVM, “Writing an LLVM pass.” Available at `http://releases.llvm.org/5.0.1/docs/WritingAnLLVMPass.html`.

[71] J. Choi, Ruo Long Lian, S. Brown, and J. Anderson, “A unified software approach to specify pipeline and spatial parallelism in FPGA hardware,” in *2016 IEEE 27th*

International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 75–82, July 2016.

[72] LLVM, “LLVM Alias Analysis Infrastructure.” Available at `https://llvm.org/docs/AliasAnalysis.html`.

[73] L. O. Andersen, *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.

[74] C. H. Gebotys and M. I. Elmasry, *Optimal VLSI Architectural Synthesis: Area, Performance and Testability*. Norwell, MA, USA: Kluwer Academic Publishers, 1992.

[75] P. G. Paulin and J. P. Knight, “Force-directed scheduling for the behavioral synthesis of ASICs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, pp. 661–679, June 1989.

[76] A. C. Parker, J. T. Pizarro, and M. Mlinar, “MAHA: A Program for Datapath Synthesis,” in *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, DAC ’86, (Piscataway, NJ, USA), pp. 461–466, IEEE Press, 1986.

[77] A. Canis, S. D. Brown, and J. H. Anderson, “Modulo SDC scheduling with recurrence minimization in high-level synthesis,” in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, Sep. 2014.

[78] J. Cong and Zhiru Zhang, “An efficient and versatile scheduling algorithm based on SDC formulation,” in *2006 43rd ACM/IEEE Design Automation Conference*, pp. 433–438, July 2006.

[79] B. R. Rau and C. D. Glaeser, “Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing,” vol. 12, (New York, NY, USA), pp. 183–198, ACM, Dec. 1981.

[80] Z. Zhang and B. Liu, “SDC-based modulo scheduling for pipeline synthesis,” in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 211–218, Nov 2013.

[81] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli, “Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It,” in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’15, (New York, NY, USA), pp. 209–220, ACM, 2015.

[82] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, "CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency," *J. ACM*, vol. 60, pp. 22:1–22:50, June 2013.

[83] J. Alglave, L. Maranget, and M. Tautschnig, "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory," *ACM Trans. Program. Lang. Syst.*, vol. 36, pp. 7:1–7:74, July 2014.

[84] J. Wickerson, M. Batty, B. M. Beckmann, and A. F. Donaldson, "Remote-scope Promotion: Clarified, Rectified, and Verified," vol. 50, (New York, NY, USA), pp. 731–747, ACM, Oct. 2015.

[85] L. Lamport, "How to make a correct multiprocess program execute correctly on a multiprocessor," *IEEE Transactions on Computers*, vol. 46, pp. 779–782, July 1997.

[86] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in Weak Memory Models," in *Proceedings of the 22nd International Conference on Computer Aided Verification*, CAV'10, (Berlin, Heidelberg), pp. 258–272, Springer-Verlag, 2010.

[87] R. P. Case and A. Padegs, "Architecture of the IBM System/370," *Commun. ACM*, vol. 21, pp. 73–96, Jan. 1978.

[88] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen, "X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors," *Commun. ACM*, vol. 53, pp. 89–97, July 2010.

[89] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," vol. 18, (New York, NY, USA), pp. 15–26, ACM, May 1990.

[90] R. B. Garner, A. Agrawal, F. Briggs, E. W. Brown, D. Hough, B. Joy, S. Kleiman, S. Muchnick, M. Namjoo, D. Patterson, J. Pendleton, and R. Tuck, "The scalable processor architecture (SPARC)," in *Digest of Papers. COMPCON Spring 88 Thirty-Third IEEE Computer Society International Conference*, pp. 278–283, Feb 1988.

[91] M. Dubois, C. Scheurich, and F. Briggs, "Memory Access Buffering in Multiprocessors," vol. 14, (New York, NY, USA), pp. 434–442, ACM, May 1986.

[92] S. V. Adve and M. D. Hill, "Weak ordering: a new definition," in *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pp. 2–14, IEEE, 1990.

[93] Alpha Architecture Committee and others, *Alpha architecture reference manual*, 2014.

[94] SPARC International Inc and David L Weaver, *The SPARC architecture manual*, 1994.

[95] F. Corella, J. M. Stone, and C. M. Barton, *A formal specification of the PowerPC shared memory architecture*, 1993.

[96] W. Pugh, "The Java memory model is fatally flawed," *Concurrency: Practice and Experience*, vol. 12, no. 6, pp. 445–455, 2000.

[97] J. Manson, W. Pugh, and S. V. Adve, "The Java Memory Model," in *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, (New York, NY, USA), pp. 378–391, ACM, 2005.

[98] H.-J. Boehm, "Threads Cannot Be Implemented As a Library," *SIGPLAN Not.*, vol. 40, pp. 261–268, June 2005.

[99] H.-J. Boehm and S. V. Adve, "Foundations of the C++ Concurrency Memory Model," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, (New York, NY, USA), pp. 68–78, ACM, 2008.

[100] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing C++ Concurrency," vol. 46, (New York, NY, USA), pp. 55–66, ACM, Jan. 2011.

[101] ISO/IEC, "Programming languages – C," ISO 9899:2011, 2011.

[102] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding POWER Multiprocessors," pp. 175–186, 2011.

[103] Arm Holdings, *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, 2014.

[104] J. Ševčík, "Safe Optimisations for Shared-memory Concurrent Programs," *SIGPLAN Not.*, vol. 46, pp. 306–316, June 2011.

[105] R. Morisset, P. Pawan, and F. Zappa Nardelli, "Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, (New York, NY, USA), pp. 187–196, ACM, 2013.

[106] M. Batty, A. F. Donaldson, and J. Wickerson, "Overhauling SC Atomics in C11 and OpenCL," *SIGPLAN Not.*, vol. 51, pp. 634–648, Jan. 2016.

[107] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides, "Automatically Comparing Memory Consistency Models," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, (New York, NY, USA), pp. 190–204, ACM, 2017.

[108] M. Batty, M. Dodds, and A. Gotsman, "Library Abstraction for C/C++ Concurrency," *SIGPLAN Not.*, vol. 48, pp. 235–248, Jan. 2013.

[109] B. Norris and B. Demsky, "CDSchecker: Checking Concurrent Data Structures Written with C/C++ Atomics," vol. 48, (New York, NY, USA), pp. 131–150, ACM, Oct. 2013.

[110] A. Munshi, "The OpenCL specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*, pp. 1–314, Aug 2009.

[111] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "GPU Concurrency: Weak Behaviours and Programming Assumptions," *SIGPLAN Not.*, vol. 50, pp. 577–591, Mar. 2015.

[112] D. Alistarh, K. Censor-Hillel, and N. Shavit, "Are Lock-Free Concurrent Algorithms Practically Wait-Free?," *J. ACM*, vol. 63, pp. 31:1–31:20, Sept. 2016.

[113] V. Gramoli, "More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms," vol. 50, (New York, NY, USA), pp. 1–10, ACM, Jan. 2015.

[114] "Liblfds: A library of lock-free data structures." Available at `https://liblfds.org`.

[115] B. Schling, "The Boost C++ Libraries," XML Press, 2011.

[116] G. Navarro and F. Claude, "libcds: Compact data structures library," 2004. Available at `http://libcds.sourceforge.net/doc/cds-api/index.html`.

[117] K. Hedström, "Lock-Free Single-Producer-Single-Consumer Circular Queue." `bit.ly/2dbr8IK`, 2014.

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

[119] M. M. Michael and M. L. Scott, "Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms," in *Proceedings of the Fifteenth Annual*

ACM Symposium on Principles of Distributed Computing, PODC '96, (New York, NY, USA), pp. 267–275, ACM, 1996.

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

[121] D. Jackson, *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2012.

[122] L. Maranget, S. Sarkar, and P. Sewell, "A Tutorial Introduction to the ARM and POWER Relaxed Memory Models." `bit.ly/2dbpUNu`, October 2012.

[123] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W.-m. W. Hwu, "Characterizing the Impact of Predicated Execution on Branch Prediction," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, (New York, NY, USA), pp. 217–227, ACM, 1994.

[124] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, "Repairing Sequential Consistency in C/C++11," *SIGPLAN Not.*, vol. 52, pp. 618–632, June 2017.

[125] D. Shasha and M. Snir, "Efficient and Correct Execution of Parallel Programs That Share Memory," *ACM Trans. Program. Lang. Syst.*, vol. 10, pp. 282–312, Apr. 1988.

[126] 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 *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 504–509, June 2011.

[127] P. B. Hansen, E. W. Dijkstra, and C. A. R. Hoare, "The Origins of Concurrent Programming: From Semaphores to Remote Procedure Calls," Berlin, Heidelberg: Springer-Verlag, 2002.

[128] P. Li and L.-N. Pouchet, "Throughput optimization for high-level synthesis using resource constraints," in *Int. Workshop on Polyhedral Compilation Techniques (IMPACT14)*, 2014.

[129] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Acm Sigplan Notices*, vol. 43, pp. 101–113, ACM, 2008.

[130] J. Liu, *Parametric polyhedral optimisation for high-level synthesis.* PhD thesis, Imperial College London, 2017.

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

[132] S. Brin and L. Page, "The Anatomy of a Large-scale Hypertextual Web Search Engine," *Comput. Netw. ISDN Syst.*, vol. 30, pp. 107–117, Apr. 1998.

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

[134] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A High-performance Graph Processing Library on the GPU," *SIGPLAN Not.*, vol. 50, pp. 265–266, Jan. 2015.

[135] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: Vertex-centric Graph Processing on GPUs," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, (New York, NY, USA), pp. 239–252, ACM, 2014.

[136] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu, "Efficient PageRank and SpMV Computation on AMD GPUs," in *2010 39th International Conference on Parallel Processing*, pp. 81–89, Sep. 2010.

[137] 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)*, pp. 44–54, Oct 2009.

[138] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[139] Y. Saad, *Iterative Methods for Sparse Linear Systems.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2nd ed., 2003.

[140] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, Dec. 2011.

[141] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *J. ACM*, vol. 46, pp. 720–748, Sept. 1999.

[142] D. Cederman and P. Tsigas, "Dynamic Load Balancing Using Work-Stealing," in *GPU Computing Gems*, Elsevier, 2012.

[143] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread Scheduling for Multiprogrammed Multiprocessors," in *Proceedings of the Tenth Annual ACM Symposium on*

Parallel Algorithms and Architectures, SPAA '98, (New York, NY, USA), pp. 119–129, ACM, 1998.

[144] 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*, PPoPP '13, (New York, NY, USA), pp. 69–80, ACM, 2013.

[145] N. Ramanathan, J. Wickerson, F. Winterstein, and G. A. Constantinides, "A Case for Work-stealing on FPGAs with OpenCL Atomics," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, (New York, NY, USA), pp. 48–53, ACM, 2016.

A. Data structure operations

Listing A.1: buffer

```
void push(bool *success, int elem, int * array, atomic_int *tail, atomic_int *head){
  int current_tail = atomic_load_explicit(tail, relaxed); ❶
  int current_head = atomic_load_explicit(head, acquire); ❷
  int next_tail = increment(current_tail);

  //Checking if buffer not full, before pushing elem and updating tail.
  *success = (next_tail != current_head)
    if(*success){
      array[current_tail] = elem; ❸
      atomic_store_explicit(tail, next_tail, release); ❹
    }
}

void pop(bool *success, int *elem, int *array, atomic_int *tail, atomic_int *head){
  int current_head = atomic_load_explicit(head, relaxed); ❺
  int current_tail = atomic_load_explicit(tail, acquire); ❻
  int next_head = increment(current_head);

  // Checking if buffer is not empty, before popping elem and updating head.
  *success = (current_head != current_tail);
  if(*success) {
    *elem = array[current_head]; ❼
    atomic_store_explicit(head, next_head, release); ❽
  }
}
```

Listing A.2: stack

```c
void push(bool *success, unsigned int node, unsigned int val,
    pointer_t* top, pointer_t *nodes_next, volatile unsigned *nodes_value) {
  pointer oldTop, newTop;
  oldTop = atomic_load_explicit(top, acquire); ❶
  nodes_value[node] = val; ❷
  atomic_store_explicit(&nodes_next[node], oldTop, relaxed); ❸
  newTop = MAKE_POINTER(node, (get_count(oldTop) + 1));
  *success =  atomic_compare_exchange_strong_explicit(top, &oldTop, newTop,
      release, relaxed); ❹
}

void pop(bool *success, unsigned int *retVal, pointer_t* top,
    pointer_t *nodes_next, volatile unsigned *nodes_value) {
  pointer oldTop, newTop, next;
  oldTop = atomic_load_explicit(top, acquire); ❺
  next = atomic_load_explicit(&nodes_next[get_ptr(oldTop)], relaxed); ❻
  if (get_ptr(oldTop) != 0){
    newTop = MAKE_POINTER(get_ptr(next), (get_count(oldTop) + 1));
    *success = atomic_compare_exchange_strong_explicit(top, &oldTop, newTop,
        release, relaxed); ❼
    if(*success){
      *retVal = nodes_value[get_ptr(oldTop)]; ❽
    }
  }
}
```

Listing A.3: queue

```
void enqueue(unsigned int val, bool *success, unsigned int node, pointer_t* head, pointer_t* tail, pointer_t* nodes_next, volatile unsigned int * nodes_value) {
  pointer enq_tail, next, tmp;
  *success = false;

  //Allocating new node
  nodes_value[node] = val;
  tmp = atomic_load_explicit(&nodes_next[node], relaxed);
  set_ptr(&tmp, 0); // NULL
  atomic_store_explicit(&nodes_next[node], tmp, relaxed);

  // check if tail is consistent
  enq_tail = atomic_load_explicit(tail, acquire); ❶
  next = atomic_load_explicit(&nodes_next[get_ptr(enq_tail)], acquire);❷
  // Check if tail and next are consistent
  bool consistent_tail = enq_tail==atomic_load_explicit(tail, relaxed);❸

  if(consistent_tail){
    // Was tail pointing to last node?
    if (get_ptr(next) == 0) {
      // Try o link node at the end of the linked list
      pointer value = MAKE_POINTER(node, (get_count(next) + 1));
      *success = atomic_compare_exchange_strong_explicit( &nodes_next[get_ptr(enq_tail)], &next, value, release, release);❹
    }
    else {
      // Swing tail to next node
      unsigned int ptr = get_ptr(atomic_load_explicit(&nodes_next[get_ptr(enq_tail)], acquire));❺
      pointer value = MAKE_POINTER(ptr, (get_count(enq_tail) + 1));
      atomic_compare_exchange_strong_explicit(tail, &enq_tail, value, release, release);❻
    }
  }
  // Swing tail to inserted node.
  if(*success){
    atomic_compare_exchange_strong_explicit(tail, &enq_tail, MAKE_POINTER(node, (get_count(enq_tail) + 1)), release, release);❼
  }
}
```

```
void dequeue(unsigned int *retVal, bool *success, pointer* free, pointer_t* head, pointer_t* tail, pointer_t *nodes_next, volatile unsigned int * nodes_value) {
  pointer deq_head, deq_tail, next, val;

  *success = false;
  deq_head = atomic_load_explicit(head, acquire);❽
  next = atomic_load_explicit(&nodes_next[get_ptr(deq_head)], acquire);❾
  bool consistent_head = atomic_load_explicit(head, relaxed) == deq_head;❿

  // Is head consistent?
  if(consistent_head){
    // Is queue empty or falling behind?
    deq_tail = atomic_load_explicit(tail, relaxed); ⓫
    if(get_ptr(deq_tail) == get_ptr(deq_head)){
      // Is queue not empty?
      if(get_ptr(next) != 0){
        atomic_compare_exchange_strong_explicit(tail, &deq_tail, MAKE_POINTER(get_ptr(next), (get_count(deq_tail) + 1)), release, release); ⓬
      }
    }
    else {
      // Try to swing Head to next node
      val = nodes_value[get_ptr(next)]; ⓭
      pointer value = MAKE_POINTER(get_ptr(next), (get_count(deq_head)+1));
      *success = atomic_compare_exchange_strong_explicit(head, &deq_head, value, release, release); ⓮
    }
  }
}
```

B. Alloy model files

We present our Alloy model file used to verify our methods of Chapters 3 and 4 to ensure their correctness, since our methods need to support weak atomics that are known to be complex and to give rise to subtle bugs.

Listing B.1: Alloy model file for our constraints of Chapters 3 and 4

```
////////////////////////////
// Relational expressions //
////////////////////////////
open util/relation

sig E {} // events

pred is_empty[r : E -> E] { no r }

pred is_acyclic[r : E -> E] { acyclic[r, E] }

fun sq[s : set E] : E -> E { s -> s }

fun imm[r : E -> E] : E -> E { r - (r.^r) }

// reflexive closure (the ?-operator in Herd)
fun rc[r : E -> E] : E -> E { r + (E <: iden) }

// lift a set to a relation (the [_] operator in Herd)
fun stor[s : set E] : E -> E { s <: iden }

pred is_equivalence[r : E -> E, s : set E] {
  equivalence[r,s]
  r in s->s
}
pred strict_partial_order[r:E->E] {
  is_acyclic[r]
  transitive[r]
}
```

```
////////////////////
// C11 executions //
////////////////////
sig Exec {
  EV : set E,           // domain of all events
  W, R, F : set E,      // writes, reads, fences
  A : set E,            // atomic events
  Acq, Rel, SC : set E, // acquire, release, sc events
  po : E -> E,          // program order (aka sequenced-before)
  cd : E -> E,          // control dependencies
  rmw : E -> E,         // linking the read and write of an RMW
  sthd : E -> E,        // same thread (E.R.)
  sloc : E -> E,        // same location (partial E.R.)
}{
  // EV captures all and only the events involved
  W + R + F = EV

  // fence, read, and write events are disjoint
  disj [W, R, F]

  // A subset of the events are 'atomic'
  A in EV

  // acquires, releases, and SC operations are all atomic
  Acq + Rel + SC in A

  // Fences are atomic
  F in A

  // The components of an RMW operation are both atomic
  rmw in A->A

  // SC reads also have acquire semantics
  (R & SC) in Acq

  // Only reads and fences can acquire
  Acq in (R + F)

  // SC writes also have release semantics
  (W & SC) in Rel

  // only writes and fences can release
  Rel in (W + F)

  // SC fences can acquire and release
  (F & SC) in (Acq & Rel)

  // RMW operations cannot be SC+nonSC
  no rmw & (((EV-SC) -> SC) + (SC -> (EV-SC)))

  // program-order is intra-thread
  po in sthd

  // control dependencies are from reads
  cd in po & (R -> EV)

  // program-order is acyclic and transitive
  strict_partial_order[po]

  // Assume po is total within each thread
  sthd - iden in po + ~po

  // rmw links a read to a write, in immediate program order, on the same
      location
  rmw in (R -> W) & imm[po] & sloc

  // sthd is an equivalence relation among all events
  is_equivalence[sthd, EV]

  // sloc is an equivalence relation among reads and writes
  is_equivalence[sloc, R + W]

}
// this predicate holds if (X,rf,co) is a well-formed execution
pred wf_exec[X:Exec, rf,mo:E->E] {

  // reads-from connects writes to reads, with each read corresponding
  // to at most one write
  rf in X.W lone -> X.R

  // reads-from connects events on the same location
  rf in X.sloc

  // modification-order is acyclic and transitive
  strict_partial_order[mo]

  // mo is a union, over all locations x, of strict total orders
  // on writes to x
  (mo + ~mo) = (X.W -> X.W) & X.sloc - iden
}

////////////////////////////////////////////////////////////
// The C11 memory model, following Lahav et al. PLDI'17 //
////////////////////////////////////////////////////////////
fun fpo [X:Exec, rf,mo:E->E] : E->E {

  (stor[X.F]) . (X.po)
}
fun pof [X:Exec, rf,mo:E->E] : E->E {
  (X.po) . (stor[X.F])
}
fun rb [X:Exec, rf,mo:E->E] : E->E {
  let allRW = (stor[X.R]) . (X.sloc) . (stor[X.W]) |
  allRW - ~rf.*~mo
}
fun eco [X:Exec, rf,mo:E->E] : E->E {
  ^(rf + mo + rb[X,rf,mo])
}
fun rs [X:Exec, rf,mo:E->E] : E->E {
  (stor[X.W]) . (rc[X.po & X.sloc]) . (stor[X.W & X.A]) . *(rf . (X.rmw))
}
fun sw [X:Exec, rf,mo:E->E] : E->E {
  (stor[X.Rel]) . (rc[fpo[X,rf,mo]]) . (rs[X,rf,mo]) . rf . (stor[X.R & X.A]) . (
      rc[pof[X,rf,mo]]) . (stor[X.Acq])
}
fun hb [X:Exec, rf,mo:E->E] : E->E {
  ^(X.po + sw[X,rf,mo])
}
fun hbloc [X:Exec, rf,mo:E->E] : E->E {
  hb[X,rf,mo] & X.sloc
}
fun fhb [X:Exec, rf,mo:E->E] : E->E {
  (stor[X.F]) . (hb[X,rf,mo])
}
fun hbf [X:Exec, rf,mo:E->E] : E->E {
  (hb[X,rf,mo]) . (stor[X.F])
```

```
}
fun podloc [X:Exec, rf,mo:E->E] : E->E {
  X.po - X.sloc
}
fun scb [X:Exec, rf,mo:E->E] : E->E {
  X.po + ((podloc[X,rf,mo]) . (hb[X,rf,mo]) . (podloc[X,rf,mo])) +
  hbloc[X,rf,mo] + mo + rb[X,rf,mo]
}
fun pscb [X:Exec, rf,mo:E->E] : E->E {
  (stor[X.SC]) . (rc[fhb[X,rf,mo]]) . (scb[X,rf,mo]) . (rc[hbf[X,rf,mo]]) . (stor
      [X.SC])
}
fun pscf [X:Exec, rf,mo:E->E] : E->E {
  (stor[X.SC & X.F]) . (hb[X,rf,mo] + ((hb[X,rf,mo]) . (eco[X,rf,mo]) . (hb[X,rf,
      mo]))) . (stor[X.SC & X.F])
}
fun psc [X:Exec, rf,mo:E->E] : E->E {
  pscb[X,rf,mo] + pscf[X,rf,mo]
}
pred Coherence [X:Exec, rf,mo:E->E] {
  irreflexive[(hb[X,rf,mo]) . (rc[eco[X,rf,mo]])]
}
pred Atomicity [X:Exec, rf,mo:E->E] {
  is_empty[X.rmw & ((rb[X,rf,mo]) . mo)]
}
pred SeqCst [X:Exec, rf,mo:E->E] {
  is_acyclic[psc[X,rf,mo]]
}
pred NoThinAir [X:Exec, rf,mo:E->E] {
  is_acyclic[X.po + rf]
}
fun cnf [X:Exec, rf,mo:E->E] : E->E {
  ((X.W -> X.R) + (X.R -> X.W) + (X.W -> X.W)) & X.sloc - iden
}
fun dr [X:Exec, rf,mo:E->E] : E->E {
  cnf[X,rf,mo] - (X.A -> X.A) - hb[X,rf,mo] - ~(hb[X,rf,mo])
}
pred DataRace [X:Exec, rf,mo:E->E] {
  is_empty[dr[X,rf,mo]]
}
pred Forced_Mo[X:Exec, rf,mo:E->E] {
  (imm[mo]) . (imm[mo]) . ~(imm[mo]) in
```

```
    (rc[rf]) . (rc[(X.po) . (rc[~(rf)])])
}
fun cde [X:Exec, rf,mo:E->E] : E->E {
  (*((rf - X.sthd) + X.cd)) . (X.cd)
}
fun drs [X:Exec, rf,mo:E->E] : E->E {
  (rs[X,rf,mo]) - ((stor[X.R]) . ((X.EV -> X.EV) - (cde[X,rf,mo])))
}
fun dsw [X:Exec, rf,mo:E->E] : E->E {
  (sw[X,rf,mo]) &
  ((((rc[fpo[X,rf,mo]]) . (stor[X.Rel]) . (rc[drs[X,rf,mo]])) -
    (((X.EV -> X.EV) - X.cd) . ((X.EV -> X.EV) - cde[X,rf,mo]))) . rf)
}
fun dhb [X:Exec, rf,mo:E->E] : E->E {
  (rc[X.po]) . *( (dsw[X,rf,mo]) . (X.cd) )
}
fun pdr [X:Exec, rf,mo:E->E] : E->E {
  cnf[X,rf,mo] - (X.A -> X.A)
}
pred Dead_Pdr [X:Exec, rf,mo:E->E] {
  pdr[X,rf,mo] in dhb[X,rf,mo] + ~(dhb[X,rf,mo])
}
pred consistent[X:Exec, rf,mo:E->E] {
  Coherence[X,rf,mo]
  Atomicity[X,rf,mo]
  SeqCst[X,rf,mo]
  //NoThinAir[X,rf,mo] // omit this unofficial axiom for now
}
pred racefree[X:Exec, rf,mo:E->E] {
  DataRace[X,rf,mo]
}
pred dead[X:Exec, rf,mo:E->E] {
  Forced_Mo[X,rf,mo]
  Dead_Pdr[X,rf,mo]
}

// this predicate holds if (X,rf,mo) is forbidden by C11
pred forbidden_by_C11[X:Exec, rf,mo:E->E] {

  // execution is dead (so its litmus test is not racy)
  dead[X,rf,mo]

  // execution is inconsistent in C11 memory model
  not(consistent[X,rf,mo])
}

/////////////////////////////////////////////////////////////
// Our constraints for Chapter 3                           //
/////////////////////////////////////////////////////////////

fun rules_chap3[X:Exec] : E->E {
  // same-location WAR, WAW, and RAW, plus RAR for atomics
  ((X.po & X.sloc) - ((X.R -> X.R) - (X.A -> X.A)))
  +
  // acquires can't move later
  (stor[(X.Acq + X.SC) & (X.R + X.W)]) . (X.po) . (stor[X.R + X.W])
  +
  // releases can't move earlier
  (stor[(X.R + X.W)]) . (X.po) . (stor[(X.Rel + X.SC) & (X.R + X.W)])
  +
  // a read cannot switch with a later read/write if they are
  // separated by an acquire-fence
  (stor[X.R]) . (X.po) . (stor[X.Acq & X.F]) . (X.po) . (stor[X.R + X.W])
  +
  // a write cannot switch with an earlier read/write if they are
  // separated by a release-fence
  (stor[X.R + X.W]) . (X.po) . (stor[X.Rel & X.F]) . (X.po) . (stor[X.W])
  +
  / a read/write cannot switch with another read/write if they are
  // separated by an SC fence
  (stor[X.R + X.W]) . (X.po) . (stor[X.SC & X.F]) . (X.po) . (stor[X.R + X.W])
}

pred wf_ppo_chap3[X:Exec, mem_weak_fence: E->E] {
  all e1,e2 : E |
    (e1->e2) in rules_chap3[X] and
    (e1->e2) in mem_weak_fence
}

pred find_bug_chap3 [X:Exec, mem_weak_fence:E->E] {
  // ppo includes (at least) all the edges that our algorithm says it must
  wf_ppo_chap3[X,mem_weak_fence]

  some rf,mo : E->E {
    //rf and mo are valid reads-from and modification orders
    wf_exec[X,rf,mo]

    // Execution is forbidden in software...
    forbidden_by_C11[X,rf,mo]

    // ... but is nonetheless allowed by a valid schedule.
    is_acyclic[rf + mo + rb[X,rf,mo] + mem_weak_fence + X.cd]
    Atomicity[X,rf,mo]
  }
}
run find_bug_chap3 for exactly 1 Exec, 3 E expect 0  // 170 ms
run find_bug_chap3 for exactly 1 Exec, 10 E expect 0 // 769 ms
run find_bug_chap3 for exactly 1 Exec, 20 E expect 0 // 5472 ms
run find_bug_chap3 for exactly 1 Exec, 30 E expect 0 // 17752 ms
run find_bug_chap3 for exactly 1 Exec, 40 E expect 0 // 57027 ms
run find_bug_chap3 for exactly 1 Exec, 50 E expect 0 // 135798 ms
run find_bug_chap3 for exactly 1 Exec, 60 E expect 0 // 280650 ms
run find_bug_chap3 for exactly 1 Exec, 70 E expect 0 // 623231 ms

//////////////////////////////////////////////////////////////
// Our constraints for Chapter 4                           //
//////////////////////////////////////////////////////////////

// Given an event-type s, this function returns all the
// s-events plus all the events that are preceded by an s-fence
fun pre_fence[X:Exec, s:Exec -> set E] : set E {
  X.s + (X.s & X.F) . (X.po)
}

// Given an event-type s, this function returns all the
// s-events plus all the events that are followed by an s-fence
fun post_fence[X:Exec, s:Exec -> set E] : set E {
  X.s + (X.po) . (X.s & X.F)
}

// Event A can synchronise with event B if ...
fun can_sync[X:Exec] : E->E {
  // ... A is a release and B is an acquire ...
  ((pre_fence[X,Rel] -> post_fence[X,Acq]) +
  // ... or A is a seq-cst and B is any atomic ...
  (pre_fence[X,SC]   -> X.A              ) +
  // ... or A is any atomic and B is a seq-cst ...
  (X.A               -> post_fence[X,SC]))
    // ... and A and B are on the same location ...
    & X.sloc
    // ... but in different threads.
    - X.sthd
}

// this predicate holds if (spo,scs) is a valid path
pred valid_path [X:Exec, spo,scs:E->E] {
  // a path is made of po-edges and can_sync-edges
  spo in X.po
  scs in can_sync[X]

  // no can_sync edge in the path is secondary.
  no (scs & (X.po) . (can_sync[X])         )
  no (scs &          (can_sync[X]) . (X.po))
  no (scs & (X.po) . (can_sync[X]) . (X.po))

  // there are no consecutive spo-edges in the path
  no spo.spo

  // there are no consecutive scs-edges in the path
  no scs.scs

  // there are no forks or joins in the path
  scs.~scs in iden
  ~scs.scs in iden
  spo.~spo in iden
  ~spo.spo in iden

  // every scs-edge is preceded and followed by a po-edge
  dom[scs] in ran[spo]
  ran[scs] in dom[spo]

  // the path never returns to a previously-visited thread
  no ((scs . *(scs + X.po)) & X.sthd)

  let source = dom[spo] - ran[scs] |
  let sink = ran[spo] - dom[scs] {

    // the path has exactly one source and exactly one sink
    one source
    one sink

    // the source and the sink share a location
    (source -> sink) in X.sloc

    // if the source and sink are both reads, they are both atomic
    source+sink in X.R implies source+sink in X.A
  }
}

// this predicate holds if ppo contains *at least* the program order
// edges that our analysis says must be preserved
pred wf_ppo_chap4 [X:Exec, ppo:E->E] {
  ppo in X.po

  // NB: using the "when" clause here seems to speed things up nicely.
  all spo, scs : E->E when valid_path[X,spo,scs] |
    spo in ppo and
    all c,d,e,f : E | (
        (c -> f) in spo and
        (c -> d) in rc[ X.po & ~scs . *~(X.po) . (can_sync[X]) ] and
        (e -> f) in rc[ X.po & (can_sync[X]) . *~(X.po) . ~scs ] and
        (d -> e) in X.po
      ) implies (d -> e) in ppo
}

///////////////////////////////////
// Top level problem statements //
//////////////////////////////////

// this predicate holds if ppo contains all the edges that it is
// supposed to contain (maybe some more besides), but the execution
// is nonetheless disallowed by C11
pred find_bug_chap4 [X:Exec, ppo:E->E] {
  // ppo includes (at least) all the edges that our algorithm says it must
  wf_ppo_chap4[X,ppo]

  // ignore fences
  some rf,mo : E->E {

    //rf and mo are valid reads-from and modification orders
    wf_exec[X,rf,mo]

    // Execution is forbidden in software...
    forbidden_by_C11[X,rf,mo]

    // ... but is nonetheless allowed by a valid schedule.
    is_acyclic[rf + mo + rb[X,rf,mo] + ppo + X.cd]
    Atomicity[X,rf,mo]
  }
}
run find_bug_chap4 for exactly 1 Exec, 3 E expect 0 // 678ms
run find_bug_chap4 for exactly 1 Exec, 4 E expect 0 // 2395ms
run find_bug_chap4 for exactly 1 Exec, 5 E expect 0 // 58432ms (58s)
run find_bug_chap4 for exactly 1 Exec, 6 E expect 0 //4402077ms (73m)

pred check_against_chap3[X:Exec] {
  // There is an edge ...
  some e1, e2 : E |

    // ... that must be preserved under our global analysis ...
    (some spo, scs : E->E | valid_path[X,spo,scs] and (e1 -> e2) in spo)

    // ... but needn't be preserved under the thread-local analysis.
    and (e1 -> e2 not in rules_chap3[X])
}
// our analysis is sometimes stronger than thread-local analysis...
run check_against_chap3 for exactly 1 Exec, 4 E expect 1

pred check_against_chap3_restricted[X:Exec] {
  check_against_chap3[X]

  // no location is accessed by both an SC atomic and a non-SC atomic
  no X.SC <: X.sloc :> (X.A - X.SC)

  // ignore fences
  no X.F

}
// ... but not if we ignore fences and assume that no location is
// accessed by both an SC atomic and a non-SC atomic.
run check_against_chap3_restricted for exactly 1 Exec, 30 E expect 0 // 763 ms
```