UNIVERSITY OF CALIFORNIA
SANTA CRUZ

PYROPE: A LATENCY-INSENSITIVE DIGITAL ARCHITECTURE TOOLCHAIN

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Haven Blake Skinner

December 2018

The Dissertation of
Haven Blake Skinner is approved:

Professor Jose Renau, Chair

Professor Martine Schlag

Professor Heiner Litz

Lori Kletzer
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Pyrope: A Latency-Insensitive Digital Architecture Toolchain

by

Haven Blake Skinner

This paper proposes a new toolchain for digital architecture development which is designed to leverage Fluid Pipelines, a variant of latency-insensitive (LI) systems developed at the UCSC Micro-Architecture (MASC) Lab. Prior work on Fluid Pipelines and other LI systems has shown that they have promising properties which can be leveraged in various ways to aid in digital architecture design, especially when developing and manipulating large digital architectures, but that there are also challenges when working with this type of system. This paper presents Pyrope, a toolchain designed for developing Fluid Pipelined digital architectures. The toolchain includes a custom hardware description language and optimizations for simulation, synthesis, and verification which leverage Fluid Pipelines. The final section of this document presents LiveSim, a live development environment built on top of Pyrope, which uses incremental compilation and hot binary reloading to update simulation results based on code changes in seconds, even for large architectures.

Acknowledgments

I would like to thank my parents, who put me on this path by raising me to value learning, the sciences, and academic research. Without their love and continuous support this PhD would not have been possible. I would also like to thank Lourdes, my girlfriend, was also a rock of support through the most difficult times in my grad school career.

My deepest gratitude also goes out to my advisor, Jose Renau, who provided the initial vision vision for Pyrope I had the privilege of developing, and who continued to support the project, and me, even when things were difficult. The same goes for the rest of the students of the MASC lab, especially Rafael Possignolo who was a close collaborator, and lead researcher for much of the early work on Fluid Pipelines.

I would finally like to thank the other friends, colleges, teachers, and many others who've influenced and inspired me over the years.

Chapter 1

Introduction

Be careful starting something you may regret.

Publilius Syrius, *Maxims*

Over the last several decades we have seen digital architectures scale from a few thousand logic units, to billions. Being able to develop systems of this size and complexity has required the continued improvement of tools which enable engineers to reason about, implement, and verify these systems.

Despite all the prior progress, however, the digital architecture design process still has significant problems, and these problems are becoming more acute as architectures continue to scale in size and complexity. Hardware design is notorious for using programming languages such as Verilog and VHDL which have limited abstraction capabilities, for example.

More importantly, the vast majority of modern architectures are designed around a paradigm called the *clock-synchronous pipeline* (further discussed in Chapter 2.1.2), which has the side effect of closely integrating the behavior of the system with its timing attributes (maximum clock frequency). This forces hardware developers to lock down design decisions such as the target clock frequency and/or the number of stages early in the design process, making it difficult to change later.

Another significant source of inefficiency in the hardware design process, which particularly affects debugging and verification, is the long latency between when a developer makes a change to the code, and when they can see the result of that change. This is due to both the time required to recompile the updated architecture, and the fact that bugs often occur after tens of thousands, hundreds of thousands, or even millions of cycles or more, and thus the simulation must be rerun to that point to access the results of that change. Sometimes developers test their systems on FPGAs rather than software simulations since those can run cycles faster, however FPGAs require more time to recompile since they require additional steps (place-and-route, uploading bit file). This paper refers to the time required between making an edit, and seeing the results, as the *edit-run-debug* (ERD) latency, that for modern digital architectures can easily be a half hour or more. Prior research has shown that long time intervals between input and feedback hinders productivity [31, 41, 57], a fact that is intuitively true to developers as well.

This paper proposes a new digital architecture design flow to address these issues, based on a variant of Latency Insensitive (LI) systems developed at the UCSC MASC lab: Fluid Pipelines. Fluid Pipelines, like all latency insensitive systems, are distinct from the traditional clock-synchronous pipeline commonly used in hardware design in that their behavior is independent of the latency between any two blocks. At the MASC lab, we have come to believe that this type of design paradigm holds a lot of promise for managing the ever increasing size and complexity of digital architectures.

In the subsequent chapters I will describe a new toolchain for developing Fluid Pipeline-based architectures. It is built around the Pyrope programming language, which is designed for implementing latency insensitive systems, and its compiler, which is designed to leverage them in a variety of ways. The language and compiler were designed to address various shortcomings with the current digital architecture design flows.

The specific contributions of the Pyrope project are summarized below.

A major motivation for the Pyrope project was to design a toolchain which could **(1) leverage LI pipeline transformations.** Chapter 4 shows how this can leveraged to aid the simulation, synthesis, and verification process. These optimizations depend on fundamental differences between Fluid Pipelines and traditional clock-synchronous circuits.

A major challenge with this project is that the Fluid Pipeline paradigm is fundamentally different from the developer perspective as well, as the system must be designed to behave consistently regardless of the latency between any two sections of the architecture. For the hardware developer, this means implementing LI behavior using simple handshake protocols and/or message passing queues, which is difficult, error prone, and not very portable. We in the MASC lab believe that the difficulty in working with these types of systems at the synthesis level is a major hindrance to their adoption.

To address this, a core part of the Pyrope toolchain is the Pyrope hardware description language, which is built on top of **(2) the Liam programming paradigm to implement**

latency-insensitive behavior implicitly (Chapter 3). Liam, which is an acronym for Latency-Insensitive Actor-Model is a custom programming paradigm which allows the programmer to describe LI behavior using the control flow structure of the source code. Though internal use by myself and other researchers, we found that this made RTL code easier to reason about in general, vastly decreased the amount of code which had to be written, without sacrificing control or tunability. Pyrope also provides a stronger type system than traditional HDLs to decrease the occurrence of overflow and other sorts of common bugs.

The Pyrope compiler is also designed for speed, to address another problem with digital architecture development discussed above. In Chapter 5, this thesis shows that for large architectures, the Pyrope compiler **(3) supports incremental compilation and hot reloading of a simulation binary.** I am not aware of any other hardware design flow that supports this. Incremental compilation and hot reloading allow Pyrope to significantly outperform Verilator (Chapter 5.6) in both compilation and simulation speed.

Chapter 5 also shows how the Pyrope compiler, along with incremental compilation, simulation binary hot reloading, and what is referred to as *checkpoint transformations* (Chapter 5.4.4), can be used to implement a **(4) digital architecture live development environment.** This requires implementing the Pyrope compiler into a Software-as-a-Service (SaaS) tool. As this paper demonstrates, Pyrope's live development environment is not just for toy designs, but can scale to even large architectures, such as as mesh of 256 RISC-V processors.

The remainder of this document is organized as follows: Chapter 2 discusses prior work on latency insensitive systems, fluid pipelines, and the Pyrope programming language. Chapter 3 focuses on Liam, demonstrating how Pyrope uses it to implement fluid behavior, followed by a detailed discussion of its implementation. Chapter 4 discusses pipeline transformations, how they are integrated into the compilation flow, and how they can be leveraged to aid in various parts of the digital architecture design process. Chapter 5 discusses the internal design of the Pyrope compiler and the LiveSim live development environment. That chapter also evaluates the viability of LiveSim to provide the response times necessary to deliver a live development experience, even with large architectures, and also compares the performance of the Pyrope compiler to Verilator [18], an open-source Verilog compiler widely used in industry and academia. Finally, Chapter 6 offers some concluding remarks.

Chapter 2

Background

The wrath of the gods may be great, but

it certainly is slow.

Decimus Iunius Iuvenalis, *Satirae*

Since the Pyrope project is an intersection of several research topics, I divide the background into the following sections: Section 2.1 discusses Fluid Pipelines, a variation of Latency Insensitive systems on top of which the Pyrope toolchain is built. Section 2.2 discusses other programming and hardware description languages which influenced Pyrope. Finally, Section 2.3 discusses hardware simulation, beginning with some background information on the challenges in that area, and on how Fluid Pipelines can be used to address them, both in prior work and in work presented in this paper.

2.1 Fluid Pipelines and the Pyrope Project

The goal of the Pyrope project is the build a digital architecture development toolchain based around Fluid Pipelines, a variation of Latency Insensitive (LI) systems. The project was inspired by some inherent challenges to developing large system with the traditional model for digital architecture design: *the clock-synchronous pipeline*. An in-depth discussion of this can be found in Section 2.1.2. Prior work on LI systems [36] [51] [45] have suggested that this model could be a good solution to manage some of those issues. Furthermore, some foundational work conducted at UCSC [63] [64], to which I had the privilege to contribute, shows that Fluid Pipelines could make better use of pipeline transformations than previous hardware based LI implementations, adding further value to this model.

The rest of this section is organized as follows: Section 2.1.1 provides an overview of the Fluid Pipelines model, and shows how it is implemented in synthesis. Section 2.1.2 compares Fluid Pipelines to the standard model for digital architecture design, the clock-synchronous pipeline. Section 2.1.3 differentiates Fluid Pipelines from other Latency Insensitive systems. Finally, Section 2.1.4 draws on all this to discuss the motivations for the Pyrope project.

2.1.1 Fluid Pipelines

Fluid Pipelines are a variation of Latency Insensitive systems developed in the MASC lab at UCSC [63] [64]. They are designed to address many of the challenges in work with clock-synchronous pipelines, the dominant model for developing digital architecture, discussed in Section 2.1.2. Fluid Pipelines differ from the clock-synchronous paradigm in that their func-

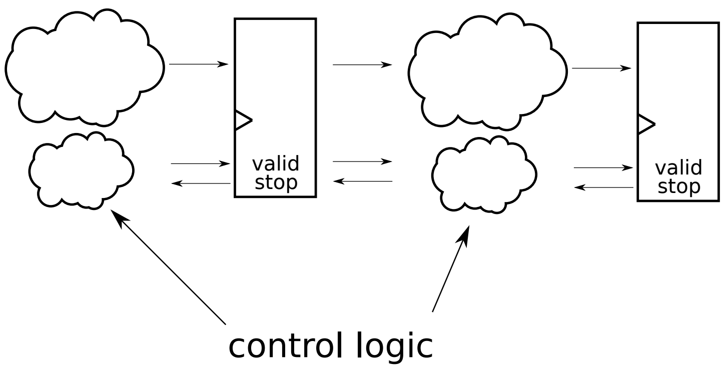


Figure 2.1: A fluid pipeline replaces synchronous registers with fluid registers, which use *valid* and *stop* signals to execute a simple handshake protocol to update its internal state.

tional behavior is not sensitive to changes in latency between any two points in the pipeline. This is further discussed in Section 2.1.2.

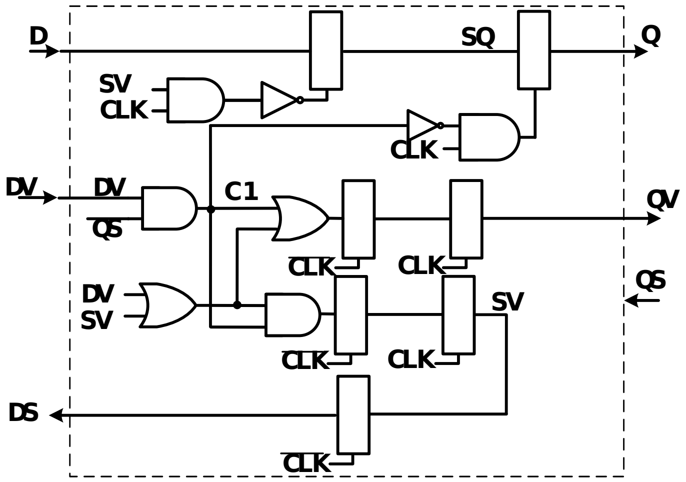


Figure 2.2: Fluid Registers are used in place of traditional clock-synchronous registers in Fluid Pipelines. They contain enough internal buffering to allow back pressure to propagate without dropping data. (Picture credit: R. Possignolo [65])

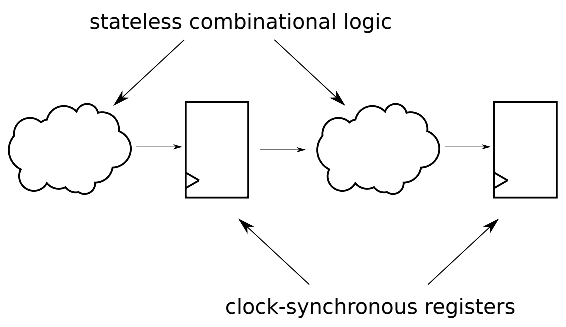


Figure 2.3: The clock-synchronous pipeline is the standard model for developing modern digital architecture. A side effect of doing so is that the system's behavior becomes closely tied to its timing.

Figure 2.1 shows a Fluid Pipeline. It is made up of combinational logic blocks divided by *fluid registers*, which have *valid* and *stop* control signals to flag valid data and handle back pressure. These control signals add LI behavior (waiting for valid data, handling back pressure without dropping packets) to the architecture.

The internal design of a fluid register is shown in Figure 2.2. Fluid registers are made using clock-synchronous registers and other control logic, implementing the equivalent of a two entry buffer. This provides exactly enough space for the *stop* signal to propagate backwards without having to drop data.

2.1.2 Clock Synchronous vs. Fluid Pipeline Architectures

The traditional model for developing digital architecture is the synchronous pipeline, shown in Figure 2.3. In a synchronous pipeline, behavior is modeled as blocks of stateless combinational logic divided between clock-synchronous register blocks. This section discusses some fundamental problems the synchronous pipeline design paradigm presents, and how the Pyrope Project hopes to address those issues with Fluid Pipelines.

One major implication of modeling a digital architecture as a clock-synchronous pipeline is that the maximum frequency at which the system can be clocked is a function of the longest propagation delay between any two registers, also referred to as the *critical path*. Creating an architecture with a fast clock speed means minimizing the critical path by carefully dividing the combinational logic between registers.

This adds many difficulties to the development process. First, it is difficult to precisely know the maximum clock speed of a circuit until after it has been implemented and the code can be analyzed by a synthesis tool. If the circuit is too slow, the only way to increase the maximum clock speed is to manually divide and/or redistribute application logic between registers. Not only is this known to be an open-ended and difficult problem [50, 70], but also makes it hard to share code between projects with different timing goals, since for systems that can run at a slower speed, removing registers saves on energy usage.

This type of tuning is critical for implementing modern digital architectures and is one of the reasons why specialized hardware description languages, which allow developers to work on the register-transistor level (RTL), are used for this purpose. Although circuits can be synthesized from more traditional programming languages, a technique referred to as *high-level synthesis*, this is rarely used in modern hardware design because the levels of abstraction make it more difficult to control what is synthesized.

Another issue with clock-synchronous pipelines is that often the clock cycle latency is what is keeping the system in sync, which makes the entire pipeline more difficult to manip-

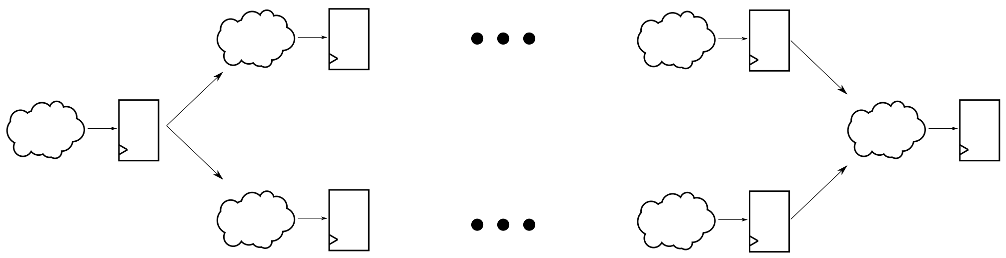
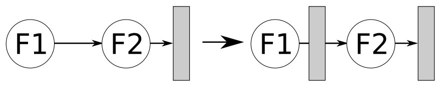


Figure 2.4: Situations which require the data path to branch often result in code that contains hidden assumptions about the latency between different sections of the pipeline, which makes the system harder to manipulate in general.

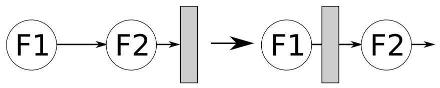
ulate. In Figure 2.4, adding or removing any block on either of the branches requires adding or removing and equivalent amount on the other branch or it could risk creating a timing error. Such architectures often have additional forward or back paths between stages in the pipeline, which often contain additional hidden assumptions about the clock latency between them. This is another factor which makes synchronous pipelines harder to work with.

Unlike clock-synchronous pipelines, Fluid Pipelines can more easily manipulated at a high level as there is no risk of creating timing errors. There is also no issue with stages being in different timing domains so long as the setup and hold conditions for the hardware components are not violated. This advantage was a motivation for early work on LI systems which proposed them as a means for more robust on-chip communication in the face of long wire lengths and arbitrary variations in latency at interconnects [36].

Another important advantage that Fluid Pipelines have over pipelines designed under the clock-synchronous paradigm is that Fluid Pipelines can be transformed more aggressively, as shown in Figure 2.5. This figure shows that fluid registers can be added, removed, or moved between blocks of combinational logic without regard to the underlying behavior of the system.



(a) Recycling



(b) Retiming

Figure 2.5: Latency insensitive pipeline transformations allow for adding or removing register blocks (Recycling) or moving logic between register blocks (Retiming), without changing behavior. (Picture credit: R. Possignolo [64])

This is possible because Fluid Pipeline transformations can leverage the fact that since behavior is independent of the latency between stages, dropping that latency to zero, or in other words merging two stages, can safely be done without breaking functionality. Likewise, increasing the latency of a region, by adding more registers, can also be done safely. Cortadella, et. al. refer to such transformations as "behavior preserving" and "correct-by-construction", where they were first proposed [45].

Leveraging transformations on Fluid Pipelines is a major motivation behind the Pyrope project. Section 2.1.3 shows that Fluid Pipelines differ from other LI systems in a way that allows developers to make better use of transformations, and Chapter 4 shows how transformations are integrated into the Pyrope compilation flow.

2.1.3 Fluid Pipelines vs. Other Latency Insensitive Systems

Fluid Pipelines are a type of LI system in that their behavior is not sensitive to changes in latency inside or outside of the architecture. There are a variety of types of LI systems including hardware implementations [36] [48] [51] [43], and distributed networks such as those built around TCP/IP. This section makes a distinction between prior hardware LI systems, fluid pipelines, and distributed networks in that prior hardware implementations provide not just a LI guarantee, but an even stronger guarantee which this thesis refers to as *sequence-invariance*. The fact that Fluid Pipelines provide latency insensitivity but not sequence-invariance allows it to avoid the performance degradation that other hardware LI systems experience with pipeline transformations.

Sequence-invariance means that not only is the system latency insensitive, but that the order of packets through any given point is also invariant with regards to latency. This is a stronger guarantee than is given by latency-insensitive systems such as TCP/IP networks, since the latency between routers can affect the path that a packet takes to its destination. The contents of the packet, however, are generally unaffected.

Early work on LI systems proposed them as a means of providing more robust on-chip communication [36]. In 2010 Cortadella et al proposed adding *elastic buffers*, their version of fluid registers, to clock-synchronous circuits as a means of transforming them while preserving their behavior [45]. For these applications, guaranteeing sequence-invariance is an implicit requirement. To accomplish this, in cases where the pipeline branches, each path must remain balanced by adding elastic buffers on the shorter paths. Furthermore, when dealing with sequential loops in the pipeline (a loop in the pipeline's data path, not to be confused by software program loops) design tools are unable to insert stages in those regions [46]. Combined, these issues have been shown to degrade overall throughput [34, 37, 51].

Sequence-invariance can also be a useful tool for verification, since if a system can be shown to have this property, the verification procedure can safely make that assumption. The Kahn Process Networks (KPN) [52] is the seminal work which formalizes sequence-invariance. It does this by modeling the nodes of the network as reading from unbounded FIFOs, with only blocking reads and non-block writes. Programmers are expressly forbidden from executing any sort of non-blocking read, or in other words, is not allowed to check if data is available on a port and make a decision based on that. Another way of thinking about this restriction is that programmers are forbidden from writing code which takes the current network conditions, i.e. the presence or absence of a packet in the input buffer, into account on any sort of decision it makes. In doing so, a KPN's behavior is guaranteed to be independent of those factors. KPNs have been applied to modeling real time systems [6], high-speed graphics processing [40], and even circuit design [43].

In the foundational work on Fluid Pipelines we argue for a more lax latency insensitive guarantee, however, as this allows the system to better leverage pipeline transformations. In "Fluid Pipelines: Elastic Circuitry Meets Out of Order Execution" [63], a paper to which I was a contributing author but not the primary one, we show that overall throughput can be improved if the system is allowed to relax the ordering guarantee. In practical terms, this means that architectures implemented with fluid pipelines must be designed to function properly even if the arrival order of some packets is changed due to pipeline transformations. Architectures designed on top of a KPN based paradigm can safely assume that the arrival order of packets will not change. We justify this in another paper to which I contributed: "Fluid Pipelines: Elastic Circuitry without Throughput Penalty" [64], where we show that removing the ordering guarantee completely eliminates the throughput degradation observed in earlier work on latency insensitive pipeline transformations.

This means the Fluid Pipelines implicitly cannot be KPNs, since their ability to fully leverage pipeline transformations is dependent on allowing packets to reorder themselves.

The distinction between Fluid Pipelines and KPNs is expressed in practical terms by the Pyrope language, discussed in detail in Chapter 3. Pyrope provides an operator ("?"), which if applied to an input port will return $true$ if that port is valid (or in other words, that a packet is present), $false$ otherwise. Used with an "if-then-else" block, this gives the developer the ability to check if that port is valid and act differently depending on the outcome. This is expressly forbidden in a KPN. An equivalent operator exists for outputs to indicate back pressure ("!"), which is likewise forbidden.

The distinction between Fluid Pipelines and KPNs is expressed in terms of synthesis blocks as well. Figure 2.6 shows the *merge* operator from "Fluid Pipelines: Elastic Circuitry Meets Out-Of-Order Execution" [63]. This is a combinational logic block with ports for two data inputs: $data_{i1}$ and $data_{i2}$, and one data output: $data_o$. Each port has corresponding valid and stop control signals. The stop signal is always in the opposite polarity of its corresponding data port.

This circuit merges the inputs $data_{i1}$ and $data_{i2}$ into output $data_o$, giving preference to $data_{i1}$, and holding either or both input with the stop signals (S_{i1} and S_{i2}) until they are used.

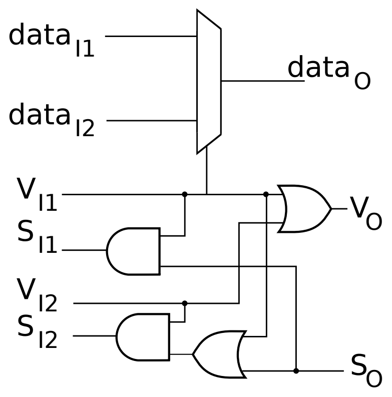


Figure 2.6: The "merge" or "fluid mux" operator selects an input based on which is valid. This can break sequence-invariance. (Picture credit: R. Possignolo [64])

More specifically, $data_o$ is connected to $data_{i1}$ if V_{i1} is true, $data_{i2}$ otherwise. V_o is true if either V_{i1} or V_{i2} is true. S_{i1} is asserted if $data_{i1}$ is valid (V_{i1}) and if the stop input, S_o is asserted (back pressure). S_{i2} if $data_{i2}$ is valid and back pressure is present (S_o), or is V_{i1} is true, since $data_{i1}$ is given priority.

Despite the complexity added by the fluid control signals, the behavior of this circuit is rather straight-forward. This circuit violates the KPN protocol nonetheless because it decides which input to select based on what is valid. This means that its behavior could be affected by the relative latencies of the input control signals.

Relaxing the ordering guarantee would certainly increase the complexity of the verification process. Chapter 4.6 proposes a verification technique that leverages Fluid Pipeline transformations, which could be helpful in alleviating some of the added complexity. In addition, speaking from anecdotal experience in making Fluid Pipelined processors, I can say that the verification complexity can also be reduced with better architecture design. Fully addressing the challenge of Fluid Pipeline verification is outside the scope of this paper, however.

Although we are not aware of any other projects strictly targetting hardware synthesis which are based on non-delay-invariant LI systems, Gorilla++ [58] is a programming language and toolchain designed for creating custom accelerators targeting streaming input processing which does have some similarities. The general flow of Gorilla++ is to compile the application into a dataflow representation, then leverage a flow similar to HLS to optimize the dataflow based on available accelerators. Most nodes in Gorilla++ are compatible with the KPN paradigm, but it also allows for nodes that can reorder packets, but may not change them, allowing it to violate the KPN paradigm for operations such as opportunistic merge. That feature was added because, as the author notes, many stream processing applications have relaxed ordering guarantees. Although the application is rather different, there are some interesting parallels.

2.1.4 Motivations for the Pyrope Project

As digital architectures continue to scale in size and complexity, so too will the difficulties associated with managing such systems under the clock-synchronous pipeline model. Latency Insensitive systems have many properties that can be helpful in managing large systems, and Fluid Pipelines expand on that model to implement architectures as a fully distributed system. The benefits of developing an architecture this way are shown in the foundational work on Fluid Pipelines [63] [64], demonstrating unrestricted use of pipeline transformations at no cost to throughput.

The early work on Fluid Pipelines also showed challenges in employing this paradigm. There can be no latency-related assumptions in the application logic, as is common when developing with clock-synchronous pipelines. More significantly, the *valid* and *stop* control signals must be implemented properly, both as inputs and outputs, for every port. When implementing Fluid Pipelines with Verilog or traditional programming languages such as C, this results in control logic being mixed in with application logic, in a manner that is difficult to separate with traditional programming abstraction constructs. Making a mistake in such control logic can result in unpredictable and difficult to find errors, such as deadlocks and invalid data being interpreted as valid data.

The goal of the Pyrope project is to bring the advantages of working with Fluid Pipelines to developers, and help manage the challenges. It employs an Actor-Model based programming paradigm designed to implement fluid behavior implicitly, allowing the developer to manipulate the behavior of the system within the bounds of the Fluid Pipeline paradigm, while sparing them from implementing the logic directly and the bugs associated with it. This is discussed in Chapter 3. The Pyrope toolchain also provides an interface to specify pipeline transformations as part of the compilation flow, discussed in Chapter 4.

2.2 Languages

The name “Pyrope” comes from the pyrope stone, which is similar in appearance to ruby. Pyrope was originally envisioned as bridging the gap between low-level HDLs and scripting languages using fluid pipelines.

Before developing the *abort-based* paradigm described in Chapter 3 we experimented with a variety of solutions to the problem of managing LI behavior from Pyrope, some which were little more than syntax sugar on top of Verilog macros, and others that were inspired by functional languages, before settling on a paradigm similar to the *Actor Model*. In addressing this problem, we had to balance the goal of simplifying the fluid pipeline development process with the need to still provide full control to the user.

This section discusses other programming and hardware description languages which are similar to, and/or have influenced Pyrope. I first discuss other HDLs in Section 2.2.1, I then present a little background on the Actor Model in Section 2.2.2, and finally some other programming languages we looked to for inspiration in Section 2.2.3.

2.2.1 Hardware Description Languages

By far, the dominant HDL in academia and industry for developing synthesizable digital architectures is Verilog. It can be thought of as the "assembly of hardware" as it provides developers with low level control to develop systems at the *register-transistor level* (RTL).

There have been a variety of projects to create better HDLs. Bluespec [61] could be considered a functional language. Circuits are inferred based on any number of "rules", one or more of which could evaluate depending on the inputs to the module. Bluespec's paradigm is perhaps most similar to Pyrope's out of all the HDLs listed here. Pyrope, however, is designed for creating Fluid Pipelines, this both restricts the programmer and provides fundamental guarantees (further discussed in Chapter 3), while Bluespec is a general purpose HDL. Another key difference is that clock-barriers in Pyrope are implicit, since they can change based on pipeline transformations, while clock-barriers in Bluespec are explicit.

Chisel [29] is an HDL developed at UC Berkeley and built on top of Scala, providing a stronger type system better abstraction capabilities than Verilog. PyMTL [59] and PyRTL [39] are built on top of Python.

Writing RTL code for synthesis is distinct from other types of programming in that the developer cannot use constructs such as loops and “malloc”-ing from main memory which cannot be translated directly into combinational logic (transistors) and finite sized storage units (registers).

Attempts to synthesize circuits from code with more traditional programming constructs is generally referred to as “high-level synthesis” (HLS). There are a variety of projects aimed at doing this [16, 35, 71]. A major limiting factor in these projects’ applicability is the importance or carefully balancing logic between register blocks in order to achieve a high clock frequency. This is more difficult when having the tool synthesize something like a loop, which will require it to infer a state machine to implement that behavior, including multiple registers and feedback paths.

Although there are similarities between Pyrope and many HLS flows, the key distinguishing feature is that HLS includes some sort of automated algorithm to optimize the resulting system for features such as frequency and area. In contrast, with Pyrope the architecture would be developed at the RTL level, enabling for high-level, but manual transformation, tuning, and optimization by leveraging latency insensitivity.

Chapter 3 discusses how Pyrope implements LI behavior, which involves inferring logic. We distinguish this from HLS flows because using Pyrope's operators, the developer has low-level control over the inferred logic network. Pyrope does not support high-level constructs such as open-ended loops.

2.2.2 The Actor Model

Pyrope's LI paradigm is based on the *Actor Model*, which is a programming paradigm first proposed in 1973 [49] as an artificial intelligence computational model. It proposed viewing the system as an interaction between *actors*, which are atomic units that communicate via message passing. Specifically, actors have the ability to: send a message, update its internal state, or create new actors.

The Actor Model has served as a concurrency model [1,11,24] and a general platform for parallel and distributed system programming [8]. It has been used in the backends of various design and simulation tools [12,21,33] and there are Actor Model libraries available for a variety of programming languages [1, 11]. The Cal Actor Language [3], is a programming language built directly on top of the actor model.

We are not aware of any HDL Actor Model libraries, however, or any HDLs which use the it as part of their design philosophy. We were attracted to it as we saw parallels between fluid pipelines and heterogeneous concurrent systems where the Actor Model is well suited to simulate. It also adapts well to an imperative programming style, that we also found appealing.

2.2.3 Other Programming Languages

There were other languages we looked at that tackled similar problems, but from which we ultimately did not draw inspiration on for Pyrope.

Synchronous programming languages like Esterel [6] and Lustre [62] are languages designed for implementing *reactive systems*, which are systems that must react to their environments within specific time constraints. Esterel, for example, defines modules in terms of input and output signals which are either “present” or “absent” at any given time, and provides constructs to do things like wait for signals and emit signals.

These types of languages are more commonly used to implement robust, time-critical systems such as flight-guidance systems, however we did see an analogy between a language based around objects with react to signals, and a fluid pipeline of stages that process packets when they arrive. There has also been prior work on synthesizing hardware with Esterel [42]. We ultimately went with a paradigm that was more similar to widely used general purpose programming languages such as C and Python.

2.3 Fluid Pipelines and Hardware Simulation

Hardware simulation is the process of implementing a digital architecture, or some part of a digital architecture, in software. Simulation is an essential tool in digital architecture design, and also often forms a bottleneck in many parts of the development process. This section provides some background on hardware simulations and why we in the MASC lab saw the potential for leveraging Fluid Pipelines to address some of the challenges in this area.

Architecture simulation can be broadly divided into two categories: cycle accurate and non-cycle accurate. Cycle-accurate means that the implementation can reproduce the architecture's behavior cycle by cycle. This is not generally possible without being able to represent the architecture's entire state, thus a cycle-accurate representation is required to *synthesize*, or generate a physical implementation of the system, and likewise an implementation which can be used for synthesis can be simulated cycle-accurately.

Cycle-accurate implementations of modern architectures are often done at the register-transistor level (RTL), and the system is generally implemented as a clock-synchronous pipeline. The exception to this is high-level synthesis (HLS), though this is generally used for fast prototyping, as HLS underperforms RTL implementations.

Non-cycle-accurate representations, cannot reproduce the architecture's behavior cycle by cycle. Examples include ESESC [28] and Gem5 [32], which simulate some part of the architecture's behavior, such as the performance of its branch predictor or estimated power usage. These types of simulations are better for early design-space exploration, such as studying the trade-offs of design decisions like the architecture's cache size, since they can be implemented more quickly than a cycle accurate simulation, and can cover ground more quickly in long testbenches.

These simulation options present a trade-off that hardware architects are forced to contend with. Cycle-accurate implementations are required for synthesis, but require more effort to implement and are slow to simulate. Non-cycle-accurate implementations cannot be used for synthesis since they do not contain complete information about the system. Likewise, there is no easy way to extract a non-cycle-accurate simulation from an RTL implementation without significant planning and infrastructure since clock-synchronous RTL implementations closely tie timing and behavior.

We in the MASC lab felt that Fluid Pipelines could be leveraged to address this trade-off. As with all LI systems, Fluid Pipelines can be more easily manipulated at a high-level since their behavior is independent of timing. This would make it easier to isolate different parts of the architecture and test them in different contexts. Conceptually simple optimizations, such as replacing a floating point unit with a software floating point operation, is simple in a LI architecture, but near-impossible in a clock-synchronous pipeline.

Fluid pipelines' ability to be transformed, changing their timing without changing their behavior, could be further leveraged to blend cycle-accurate and non-cycle-accurate simulation. Chapter 4 shows how 4 and 5-stage RISC-V CPUs can be transformed into single-stage CPUs. Compiled to C, these implementations are functionally similar to software emulators; that chapter also compares their performance against the official RISC-V reference emulator, which is written in C.

Chapter 4 also shows how transformations cause some architectural optimizations, such as branch predictors and forwarding paths, redundant, providing a speed up by simplifying the system as well, analogous to a non-cycle-accurate simulation.

These optimizations directly allow developers to trade speed for cycle-accuracy, but using the same code-base, rather than having to implement the system multiple times.

Prior work has also proposed other types of hybrid simulations for which Fluid Pipelines could also be helpful.

SMARTS [74] sampling framework combines a slow, cycle-accurate simulation and a fast, non-cycle accurate simulation, to provide faster measurements of power-usage and CPI (cycles-per-instruction) over a long testbench. With Fluid Pipeline Transformations, SMARTS could be implemented with one code base.

Another project [68] addresses the problem that not only is cycle-accurate simulation slow, but providing cycle by cycle values of every variable in the simulation is even slower, even though this is generally needed for effective debugging. The authors propose using a fast simulation which contains a minimal state representation of the system, taking checkpoints, then using the checkpoints to do a detailed simulation in parallel. This system could also be implemented with one code base with Fluid Pipelines.

Chapter 3

The Liam Programming Paradigm

You wonder why travel did not improve you? You had yourself as a companion.

Seneca the Younger, *Epistulae Morales*

This chapter discusses the Latency-Insensitive Actor-Model (Liam) paradigm, that is central to how Pyrope is used to create Fluid Pipelines. First, Section 3.1 discusses how Liam works, and shows how it can be used with code examples. Next, Section 3.2 shows how Liam is implemented by the Pyrope compiler.

3.1 Describing Fluid Behavior with Liam

3.1.1 Overview

The motivation behind Liam, and what it adds to Pyrope, is shown in Figure 2.1. We can see that the added difficulty in working with fluid pipelines, or any type of latency insensitive system, is the additional work of implementing logic for the control network to drive the *valid* and *stop* signals (or the equivalent). If one were implementing a fluid pipeline architecture in Verilog, one would have to implement all of that logic manually. Instead, Liam provides a paradigm designed to infer that network based on the structure of the code.

The way we manage this is by leveraging the Actor Model, as discussed in Section 2.2.2. In the abstract, the Actor Model defines the system in terms of actors who can affect the state of the system by either updating its internal state or sending messages to each other (external state). To be clear, this does not refer to a finite, countable, number of states as in a finite state machine, but an uncountable number of states described by the value of each actor's internal memory and the packets in its message passing queue. Liam provides a mechanism to manage these state transitions in the abstract, and guarantee that they happen atomically. As an example, if the developer's intention is that a stage will send a packet and update an internal counter to reflect that, Pyrope code can be structured to guarantee that either both of those things will happen, or neither of them will.

The Pyrope compiler compiles Liam behavior into logic networks to drive the *valid* and *stop* control signals of Fluid Registers. Liam is not specific to HDLs however, and could be adapted by other languages and toolchains.

3.1.2 Liam within a Pyrope Architecture

Pyrope architectures are divided between any number of *pipes* and *stages*. Stages contain logic and have an internal state characterized by their internal registers and their IO ports. Pipes instantiate stages, and can connect the IO ports of stage instances to each other, and to the pipe's own IO ports. They have no logic or internal state of their own.

Liam is designed to manage how data-path logic interacts with the control paths and ultimately the state of the overall system. Thus, it only applies to blocks with logic and state, referred to as *stages*, and not with *pipes*, which have neither.

As of now, all stages in Pyrope are implicitly fluid (although that will change in future versions). This means that each IO port is actually a fluid port with *valid* and *stop* control signals, and that logic to implement Liam behavior will be inferred for each of them by the compiler.

3.1.3 Inferring Fluid Behavior

```
mux :: {
  s as input logical
  a,b as input bits:32
  o as output bits:32

  if s {
    o = a
  } else {
    o = b
  }
}
```

Listing 3.1: A Pyrope implementation of a simple multiplexer, with two data inputs: a, b, one data output: o and one select input: s If instantiated as a stage, fluid pipeline control logic will be inferred for its input and output ports.

Listing 3.1 shows an implementation of a simple multiplexer in Pyrope, which routes the data from one of its input ports to its output port based on a selector input. If this block is instantiated as a stage in another block, all of its inputs and outputs will be considered fluid inputs and outputs, which will each implicitly have their own *valid* and *stop* control signals.

To describe fluid behavior for this stage, Liam creates the concept of the stage's *state*, which includes the values of the block's internal registers, the values on its output ports (and whether or not that output port is valid), and the state of each input port (whether or not the block is asserting back pressure on that port). The Pyrope paradigm provides rules, or Axioms, 1 (Atomicity), 2 (Abortion), 3 (Consumption), and 4 (Isolation), to describe when and how a stage's state is updated.

Axiom 1 (Atomicity) *A stage is a block of code that evaluates from the top down. If it completes without aborting, it updates its state. All outputs written to are valid in the new state. All registers written to are updated. All inputs read from are consumed.*

Axiom 2 (Abortion) *A stage aborts when an invalid input is read, or an output with back-pressure is written.*

Axiom 3 (Consumption) *All valid inputs are held (back pressure) until they are consumed.*

Axiom 4 (Isolation) *An aborted stage does not consume any input, generate any valid output, or make any other changes to its state.*

Liam adds the concept of stage aborting as a means of managing state. When an abort happens in a cycle, no inputs are consumed or outputs are produced. This concept only exists within the programming language, and is based on the control flow structure of the stage. When synthesized, this behavior is implemented as combinational logic to drive a fluid register's *valid* and *stop* signals. Implementing Liam in synthesis is further discussed in Section 3.2.

Considering the axioms, if s is valid and equal to 1, the control flow will lead to line 7, otherwise it will lead to line 9. If s is valid and equal to 1, and a is valid, than both those inputs will be consumed and o will equal a; likewise if s is valid and equal to 0 with regards to input b.

The mux in Listing 3.1 actually acts more like a switch, as it will it will only consume one of the data inputs, a and b, each time it produces output. The data input which is not selected, or "consumed" will be held with the stop signal.

3.1.4 Liam-Specific Operators

The Liam paradigm includes operators designed to allow the programmer to check the state of an IO port, and make a decision based on that. The Pyrope version of these operators is shown in Table 3.1. The "Valid Check" and "Stop Check" allow the developer to check if an input is valid or if an output has back pressure without triggering an abort.

As stated in the Consumption Axiom 3 (Consumption), an input is consumed if it is read and the stage does not abort that cycle. The "Keep Statement" allows the user to prevent that, keeping the input on the next cycle as well. Finally the "Consume Statement" does the opposite, marking an input as read, and triggers an abort if the input is not valid. This is syntax sugar since the same could be accomplished by just reading the variable and not using the value, but it improves readability.

Table 3.1: Liam adds some new constructs which helps the programmer manage elastic behavior abstractly.

Name	Code form	Description
Valid Check	*var*?	Checks if an input or output is valid
Stop Check	*var*!	Checks if an input or output stop flag is asserted.
Keep Statement	keep *input*	Prevents an input from being consumed.
Consume Statement	consume *input*	Explicitly consume an input.

Consider that for some application we may want this to act as a "fluid mux", and not a "fluid switch", meaning it should wait for both input ports to have valid values, then output one and drop the other. A stage with that behavior is shown in Listing 3.2. Pyrope has the syntax sugar *read*, that marks a port as having been read on that path. In a non-stage block this line has no effect. In a stage, however, the two *read* commands will force an abort unless the unselected input is also valid, and will cause that port to also be consumed when the state is updated.

By default, variables in Pyrope do not maintain their values across cycles. Those that do are marked with @. Placing the increment above the "if-else" block implements the desired behavior because if the stage fails to send a packet it will abort, preventing the increment of @*sent_ctr* from going through. Placing the increment at the bottom of the block may be more intuitive, however.

As another example, it may be the case for some application that we would want a "fluid mux" which would consume the input on the unselected port if it is valid, but ignore that port otherwise. Listing 3.3 introduces the ? operator which checks if an input port is valid, without triggering an abort. An equivalent operator exists for output ports: !, which checks for the presence of back pressure.

```
mux :: {
   s as input logical
   a,b as input bits:32
   o as output bits:32

   @sent_ctr as bits:32
   @sent_ctr++

   if s {
      o = a
      read b
   } else {
      o = b
      read a
   }
}
```

Listing 3.2: This version of the mux will only produce output once both input ports have a valid value. The other port is read and discarded. It also maintains an internal count of how many packets it sent.

```
mux :: {
  s as input logical
  a,b as input bits:32
  o as output bits:32

  if s {
    o = a
    if b? { read b }
  } else {
    o = b
    if a? { read a }
  }
}
```

Listing 3.3: This version will only consume the input on the unselected port if it is valid, and will ignore that port otherwise.

The three examples 3.1 3.2 3.3 show the general idea behind Pyrope's abort-based paradigm. This paradigm allows the developer to manage the fluid aspects of the system, in other words the presence of valid and invalid data and back pressure, using the control flow structure of the stage. We found that it came naturally to reason about a stage's behavior in sequential terms, such as: "the stage gets to this line and aborts", or: "it checks for back pressure here, then continues", even though we knew that the logic that drives the control signals is actually evaluated in parallel in the synthesized system.

This also prevents some types of bugs we encountered when manually implementing fluid pipelines, such as invalid data accidentally being interpreted as valid data or data with back pressure being overwritten due to small typos in the control logic. Pyrope will not generate a control network which would act that way, that in our experience made fluid pipelines much more predictable to work with.

These examples also show why a tool which automatically converts a synchronous circuit like a mux attached into a clock-synchronous register to a latency insensitive system will never be ideal. The LI system contains additional vectors of control: the concept of valid/invalid data and of back pressure/consumption. Those vectors of control must be available to the developer or their ability to optimize and tune the system will be limited.

3.1.5 Catching the Abort with "Try" Blocks

```
one2two :: {
  inp as bits:64 input
  o1,o2 as bits:64 output

  try {
    o1 = inp
  } else {
    o2 = inp
  }
}
```

Listing 3.4: This stage will route a valid input from "inp" to the first available output port.

Liam models a stage's abort as an exception. If it is uncaught, the stage will abort, however the abort can be avoided if it is triggered inside a "try" block. Listing 3.4 shows a simple stage that will send its input to the first available output. It does this by first attempting to read from input *inp* and write that result to output $o1$. If that fails either because *inp* is not valid or $o1$ has back pressure, the abort will be caught and the stage will attempt to write *inp* to $o2$.

Listing 3.5 shows how the try block can be used to implement a register file with two read ports and one write port. Note that the decision to put the reads into two separate "try" blocks, rather than the same one, may require some architectural level consideration. If both inputs $r1addr$ and $r2addr$ arrive on the same cycle but there's back pressure on only one output port, then it's possible for the results of the reads to appear on very different cycles. Putting them both in the same "try" block guarantees their read values will be produced on the same cycle by regfile, if that behavior is desired.

```
regfile :: {
  r1addr as bits:WORD input
  waddr as bits:WORD input
  r1 as bits:WORD output
  @data as bits:WORD count:32 register

  try {
    r1 = @data[r1addr]
  }

  try {
    r2 = @data[r2addr]
  }

  try {
    @data[waddr] = wdata
  }
}
```

Listing 3.5: The try block allows operations to be isolated, so that their abortion does not affect the rest of the stage.

Removing the "try" blocks would mean that a read could not happen unless a write was requested, and vice versa. This is likely not desirable, in fact, there's a good chance it could cause a deadlock in many processor implementations. No programming language can prevent the developer from making a mistake, though in my opinion, Pyrope's paradigm makes that sort of error easier to recognize and reason about, as opposed to having the control logic implemented manually in the same design space as the data logic. This is something I was not able to test experimentally, unfortunately.

The "try" block does not technically add any new functionality, as its behavior could be implemented with the ? and ! operators. It is useful because it frees the programmer from having to manually reason about what needs to be checked in a given sub-block to prevent the entire stage from aborting. This can make it easier to describe behavior such as the regfile in Listing 3.5, where the desired behavior is for each statement to be *tried* separately.

3.2 Compiling Liam

Pyrope's paradigm is designed to describe fluid behavior implicitly based on where registers and IO ports are read and written to in the control flow structure of the stage. This behavior can be compiled into control logic that drives the *valid* and *stop* signals of fluid registers. As stated in Section 3.1.2, this only applies to *stages*, that have logic and internal state. *Pipes* do not have this implicit logic compiled in.

This section is divided into several parts. The first, Section 3.2.1, discusses how Liam manages state in more detail. Then, Section 3.2.2, discusses deriving logic for the stage's external state, in other words its IO ports. Section 3.2.3 discusses deriving logic for a stage's internal state, meaning its registers.

The final section of this chapter, Section 3.3, discusses some of the challenges Liam presents when implementing large, internal memory blocks. Conceptually, Liam treats large and small registers the same. Due to practical challenges of implementing large memory blocks in synthesized hardware, and Liam's ability to abort a state change, however, the Pyrope compiler treats scalar and arrays registers differently, implementing special optimizations for the latter.

3.2.1 Stages and State

A stage's state is defined as the values of its internal registers (if it has any), along with the state of each of its IO ports (whether they have valid data and/or are asserting back pressure). The data at each IO port is also part of the stage's state if that data is valid. Invalid data is considered "don't care".

For descriptive purposes, this thesis refers to a stage's registers as its *internal state*, and a stage's IO ports as its *external state*, as these require different considerations to manage, as discussed in Section 3.2.2 and 3.2.3, and Section 3.3.

To provide access to the state Liam supports several types of variables, each of which could exist as scalar values or arrays:

Public: Visible outside the stage, in other words IO ports. Reading and writing to these means accessing fluid ports which could trigger an abort.

Register: These variables retain their values across cycles, making them implicitly part of the stage's internal state.

Private: These variables don't retain their values between cycles. They correspond to combinational logic in synthesized hardware.

Within the *register* variable type, Liam supports two kinds: *inline registers* and *deferred registers*. The difference is illustrated by Listings 3.6 and 3.7. It is important to note that inline versus deferred refers to how registers behave during a cycle. Inline registers act like software variables, updating after being assigned to in the control flow, while deferred registers act like hardware registers, updating on the next cycle. This is discussed in detail in Section 3.3.1. Both register types behave the same across cycles, in that they both will only commit their changes if the stage does not abort.

```
@reg1 as inline
@reg1 = 5
x = @reg1   # x = 5
```

Listing 3.6: The value of an inline register is updated after it is written, but that value will not persist to the next cycle if the stage aborts.

```
@reg1 as deferred
@reg1 = 5
x = @reg1   # x = <previous value>
```

Listing 3.7: Deferred registers are updated at the end of the cycle, assuming the stage doesn't abort.

Inline registers require a bit more consideration to implement, especially with regards to memory blocks as discussed in Section 3.3. The Liam paradigm provides support as one of the goals of the paradigm was to provide a programming interface that is as intuitive as possible. From a functionality perspective it is not necessary to support both, as any algorithm written with inline registers could be reworked to use deferred registers. Since Liam is not a programming language, but a programming paradigm which could be adopted by languages, it is left to the language designer to decide which parts of Liam to support.

3.2.2 Deriving Logic for External State

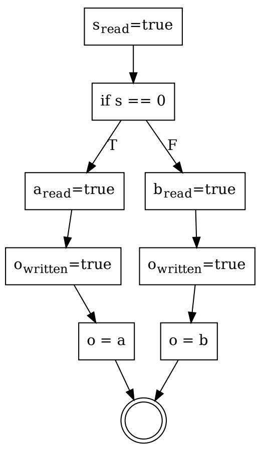


Figure 3.1: The first step to adding fluid behavior to a Pyrope block is to add flags indicating when an input or output port has been read or written. This figure shows the flags added to Listing 3.1.

Adding fluid behavior is done in several steps. First, flags are added to the control flow representation to indicate when a given input/output port or register is read or written (Figure 3.1). When this graph is converted into a dataflow graph, those flags provide us with a boolean logic network for each input/output port indicating if it is read or written to in this cycle. This is shown in Figure 3.2. Using the outputs of these logic networks, we can derive an equa-

Equation 3.1: A boolean flag indicating whether or not the stage will abort can be set with the equation below.

$$\begin{aligned} abort =& (\exists input : \neg input_{valid} \wedge input_{read}) \vee \\ & (\exists output : output_{stop} \wedge output_{written}) \end{aligned} \tag{3.1}$$

Equation 3.2: A given output is valid if the stage did not abort and it has been written.

$$output_{valid} = \neg abort \wedge output_{written} \tag{3.2}$$

tion for if the stage will abort (Equation 3.1), and equations for the output *valid* (Equation 3.2) and input *stop* (Equation 3.3) control signals for each output and input port respectively.

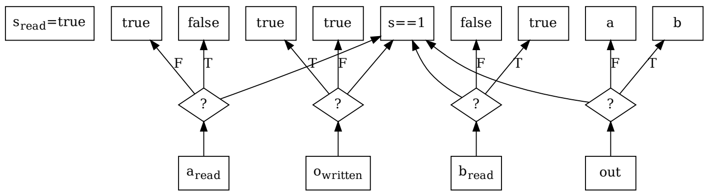


Figure 3.2: The dataflow graph generated by Listing 3.1.

Try-else blocks are implemented as shown by Figure 3.3. Both the try and the else blocks evaluate in parallel, if the try does not abort, the state update comes from that block, otherwise it comes from the else block.

Equation 3.3: A given input asserts its stop flag if it is valid and either the stage aborted, or it was not read this cycle.

$$input_{stop} = input_{valid} \land (abort \lor \neg input_{read}) \tag{3.3}$$

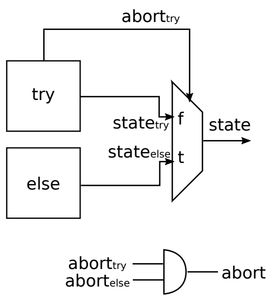


Figure 3.3: If the try block does not abort, the state update comes from that block, otherwise it comes from the else block.

Creating a dataflow representation of the logic based on the control flow representation is a standard method of converting sequential operations into parallel logic. Dataflow graphs are particularly useful for *register-transistor level* (RTL) HDLs since they do not support loops that cannot be unrolled. The dataflow graph can derive the final value of each variable in the block as an expression of its data dependencies, with all the expressions being as parallelizable as possible. Thus the control logic Pyrope generates can also be evaluated in parallel to the data path, excluding only places where the control flow branches (for example, if split by an $if-then-else$ block).

Equation 3.4: The update equation for a scalar register.

$$reg_{update} = reg_{written} \land \neg abort \tag{3.4}$$

3.2.3 Deriving Logic for Internal State

Reads and writes to a stage's internal state, also referred to as its *registers*, will never cause an abort, however writes to registers are still subject to abort rules, to prevent their state from going out of sync with the stage's external state.

The dataflow graph derived for a register will be different depending on if the register is deferred or inline, as illustrated by Listings 3.7 and 3.6. For array registers that are deferred, additional logic is required to determine if any of the indexes written to correspond to the value being read. This is discussed in Section 3.3.

The update logic for a scalar register is shown in Figure 3.4. Writes to scalar registers are marked with flags so that in the dataflow graph, for each scalar register x, there is a corresponding $x_{written}$. This, along with the *abort* flag, determine if the register is written this cycle, or if its old value remains.

3.3 Arrays and Liam

Section 3.2.3 shows how registers are updated, by building a dataflow representation of the register's updated value, and adding logic to prevent a write if the stage aborted on that cycle. Section 3.2.1 also discusses how registers can be either inline or deferred, affecting how they behave when written and read during a cycle.

There are some subtleties when extending this model to include array registers, that are discussed in this section. Section 3.3.1 describes the differences between C and Verilog arrays, and the motivation for adding these features to the language. Section 3.3.2 describes how array operations are represented internally. Finally, Section 3.3.3 discusses the array-specific optimizations that can be used to eliminate any unnecessary data duplication in the compiled code, to create a data-minimal implementation of the stage's behavior for all array registers.

3.3.1 Software Arrays and Hardware Memory Blocks

In traditional programming languages, arrays are contiguous blocks of memory that store any number of the same data-type sequentially. Each element can be accessed by index, and the semantics of operations on an array elements are generally the same as for a scalar variable.

Verilog has a feature that is similar to an array, called a memory block. A memory block declaration is shown in Listing 3.8. The numbers in the first pair of brackets indicate that each "line" of the memory block is 64-bits long, and the second number indicates that there will be 512 lines. There are similarities between memory blocks and arrays, as the line width is analogous to the array type and the line count is analogous to the array length.

```
    reg [63:0] memory_block[512];
```

Listing 3.8: A Verilog declaration for a memory block with 512, 64-bit lines. There are similarities between Verilog memory blocks and software arrays, but they are not interchangeable.

Implementing efficient hardware means carefully tuning the combinational logic networks between register blocks to minimize propagation delay. As a result, memory blocks in hardware require at least one clock cycle to read or write, sometimes more.

Another important subtlety when working with hardware memory blocks is that if there is more than one write, those writes happen in parallel. This means that multiple writes to the same index is undefined, and is generally considered bad design. This is in contrast to software arrays where writes happen in sequence and overwriting the same address is generally not a problem.

Verilog has another feature which provides behavior similar to arrays, called a *bus*. A bus is just a group of wires, numbered sequentially. As an example, Verilog provides semantics to declare a bus of 512 wires (512-bit bus), and reference them in groups of 32, treating each as a separate “element” of the “array”. Buses do not maintain their value across cycles, however, a register is required for that.

The subtle differences between software arrays and hardware memory blocks were important for two reasons. First, because with Pyrope, the goal was to provide the convenience of software arrays, while still synthesizing good quality hardware. Second, because Pyrope was designed to compile into both C++, for optimized simulation and live development, and into Verilog for synthesis. Pyrope’s internal representation had to be compatible with both.

The subtle differences between software arrays and hardware memory blocks were important for two reasons. First, deferred array registers behave like Verilog memory blocks, writes to them are modeled as taking place at the end of the cycle in parallel. As in Verilog, writing to the same location multiple times is undefined. Inline array registers, even when

Equation 3.5: The *awr* operation allows Pyrope to treat writes to array indexes no differently than scalar variables. Redundant copying can be remove with later optimizations.

$$awr(a, p_0, ...) \rightarrow a_m \tag{3.5}$$

synthesized, have logic to make them behave as software arrays. If the same index is written to multiple times, the "last" write to evaluate, from a control flow perspective, is the one that will go through. This generally requires more logic to implement, but guarantees that the compiled stage will not have undefined behavior.

3.3.2 Pyrope Internal Array Representation

To represent arrays internally, Pyrope defines a special operation: *awr*, for "array write".

The *awr* operation takes an array as its first argument, followed by one or more index-value pairs. It will return a new array that is identical to the first argument, but with the indexes specified by each pair replaced with the corresponding value.

There is one important constraint placed on the compiler, for each of the pairs passed into *awr*: the resulting array should be identical regardless of what order the pairs are written. This guarantees consistent performance whether the operation will be implemented in sequence in software, or in parallel in hardware. This constraint is referred to as the *awr-write-order-invariance* constraint.

```
array_example :: {
  @arr[addr1] = data1
  y = @arr[yaddr]
  @arr[addr2] += data2

  if data3 < 10
    @arr[addr3+1] = data3
  else
    @arr[addr3-1] = data3

  z = @arr[zaddr]
}
```

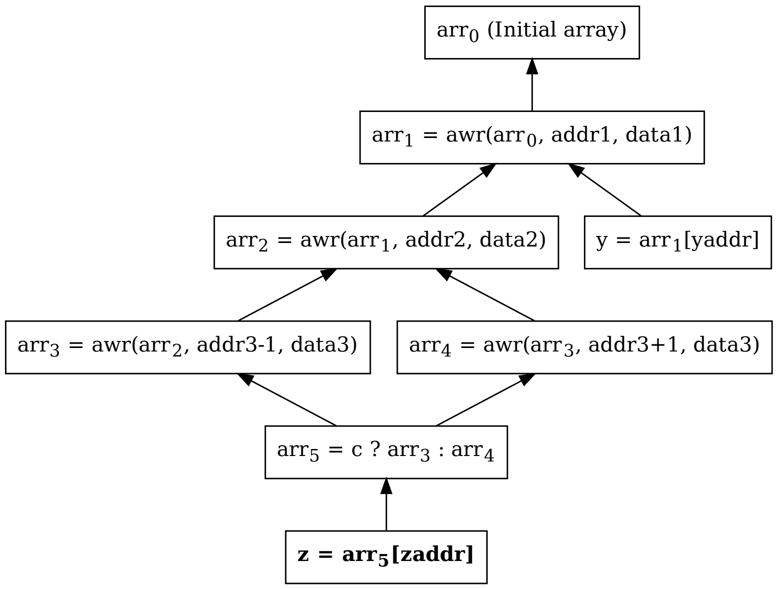
Listing 3.9: Arrays require special consideration when implementing them with Pyrope's paradigm.

Listing 3.9 shows a Pyrope stage with several array reads and writes. Figure 3.4 shows this block of code compiled into a data flow graph, using *awr* to represent the array operations. The same block of code is shown compiled with both an inline and a deferred array to demonstrate the difference. Considering each *awr* has only one index-argument pair, it is trivial to see that the *awr-write-order-invariance* constraint is maintained.

The following section lists the dataflow optimizations that can be applied to *awr* nodes to remove any redundant copying.

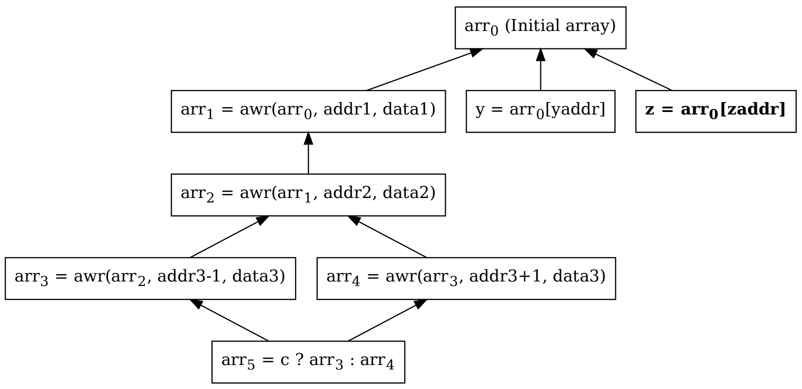
3.3.3 *awr* Optimizations

Figures 3.5, 3.6, 3.7, and 3.8 show the dataflow optimizations that are specific to *awr* operations to remove any redundant copying. Additional logic must be inferred to maintain the *awr-write-order-invariance* constraint.



arr$_0$ (Initial array)
arr$_1$ = awr(arr$_0$, addr1, data1)
arr$_2$ = awr(arr$_1$, addr2, data2)
y = arr$_1$[yaddr]
arr$_3$ = awr(arr$_2$, addr3-1, data3)
arr$_4$ = awr(arr$_3$, addr3+1, data3)
arr$_5$ = c ? arr$_3$: arr$_4$
z = arr$_5$[zaddr]

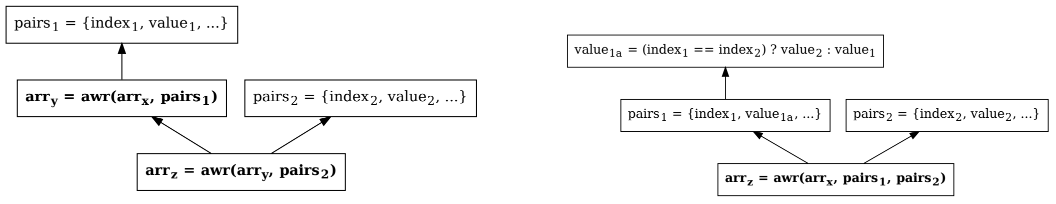
(a) Inline



arr$_0$ (Initial array)
arr$_1$ = awr(arr$_0$, addr1, data1)
y = arr$_0$[yaddr]
z = arr$_0$[zaddr]
arr$_2$ = awr(arr$_1$, addr2, data2)
arr$_3$ = awr(arr$_2$, addr3-1, data3)
arr$_4$ = awr(arr$_3$, addr3+1, data3)
arr$_5$ = c ? arr$_3$: arr$_4$

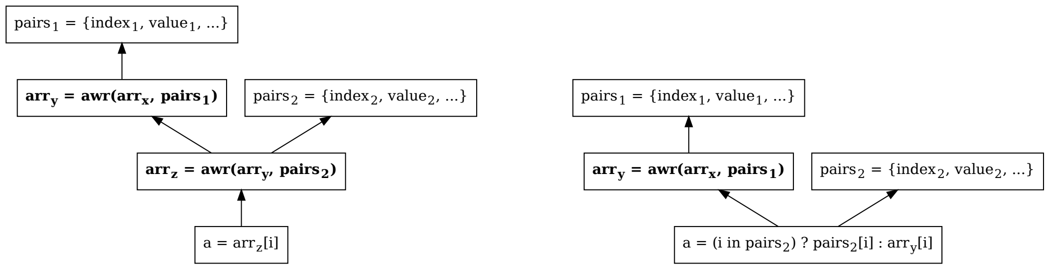
(b) Deferred

Figure 3.4: Arrays in Pyrope are either *inline* if they are updated immediately after being written, or *deferred* if they are updated at the end of the cycle. The distinction is reflected in their respective dataflow graphs; for inline arrays, reads are dependent on the latest version of the array, while deferred reads are always dependent on the version present at the beginning of the cycle.


pairs$_1$ = {index$_1$, value$_1$, ...}
arr$_y$ = awr(arr$_x$, pairs$_1$)
pairs$_2$ = {index$_2$, value$_2$, ...}
arr$_z$ = awr(arr$_y$, pairs$_2$)
value$_{1a}$ = (index$_1$ == index$_2$) ? value$_2$: value$_1$
pairs$_1$ = {index$_1$, value$_{1a}$, ...}
pairs$_2$ = {index$_2$, value$_2$, ...}
arr$_z$ = awr(arr$_x$, pairs$_1$, pairs$_2$)

(a) Before **(b)** After

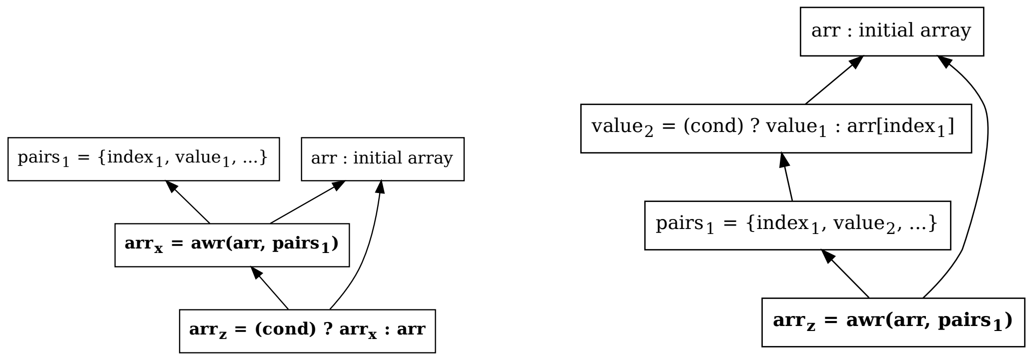
Figure 3.5: A chain of *array_writes* requires that all indexes not written be copied unmodified. This can be collapsed to a single *array_write*.


pairs$_1$ = {index$_1$, value$_1$, ...}
arr$_y$ = awr(arr$_x$, pairs$_1$)
pairs$_2$ = {index$_2$, value$_2$, ...}
arr$_z$ = awr(arr$_y$, pairs$_2$)
a = arr$_z$[i]
pairs$_1$ = {index$_1$, value$_1$, ...}
arr$_y$ = awr(arr$_x$, pairs$_1$)
pairs$_2$ = {index$_2$, value$_2$, ...}
a = (i in pairs$_2$) ? pairs$_2$[i] : arr$_y$[i]

(a) Before **(b)** After

Figure 3.6: When a read is dependent on an *array_write*, that dependency can be moved to an earlier version of the array as long as the logic to forward an updated value is implemented.

These optimizations can be continually applied until all array reads and writes are dependent on only the original version of the array, along with any number of scalar dataflow nodes. This achieves the original goal of allowing Pyrope to compile arrays that would be synthesized properly as memory blocks in hardware.

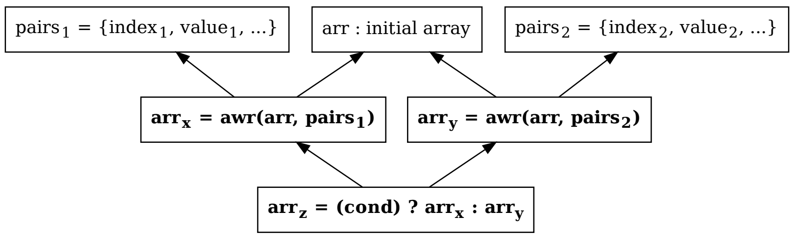

pairs$_1$ = {index$_1$, value$_1$, ...}
arr : initial array
arr$_x$ = awr(arr, pairs$_1$)
arr$_z$ = (cond) ? arr$_x$: arr
arr : initial array
value$_2$ = (cond) ? value$_1$: arr[index$_1$]
pairs$_1$ = {index$_1$, value$_2$, ...}
arr$_z$ = awr(arr, pairs$_1$)

(a) Before **(b)** After

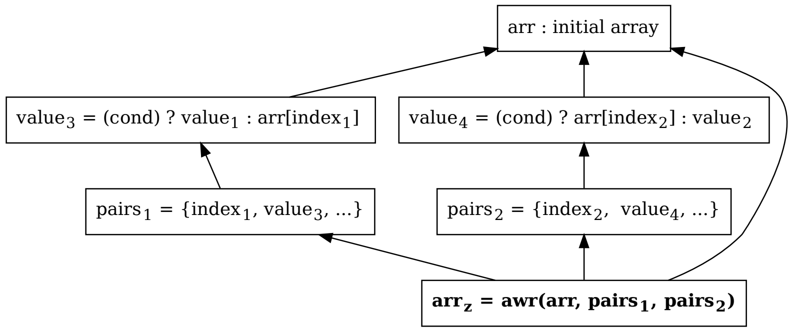
Figure 3.7: This corresponds to the case of an array being written to on one branch of an $if-else$ block. The result essentially collapses the dataflow logic network around the values being updated, eliminating unnecessary copying.

3.4 Conclusion

The Liam paradigm provides a way to add fluid behavior to blocks of code implemented as dataflow logic by adding high-level controls, in the form of Axioms (1 (Atomicity), 2 (Abortion), 3 (Consumption), 4 (Isolation)), to allow programmers to specify atomic updates to the block's state. The behavior specified by the Liam paradigm is implemented by the Pyrope language, by the Pyrope compiler, which implements Liam's behavior as combinational logic networks to drive the *valid* and *stop* control signals of fluid registers, used in fluid pipelines (introduced in in Chapter 2.1.2). This requires that the compiler generate additional logic, and requires special consideration when dealing with large memory blocks.


pairs_1 = {index_1, value_1, ...}
arr : initial array
pairs_2 = {index_2, value_2, ...}
arr_x = awr(arr, pairs_1)
arr_y = awr(arr, pairs_2)
arr_z = (cond) ? arr_x : arr_y

(a) Before


arr : initial array
value_3 = (cond) ? value_1 : arr[index_1]
value_4 = (cond) ? arr[index_2] : value_2
pairs_1 = {index_1, value_3, ...}
pairs_2 = {index_2, value_4, ...}
arr_z = awr(arr, pairs_1, pairs_2)

(b) After

Figure 3.8: This corresponds to the case of an array being written to on one branch of an $if-else$ block. The result is similar to Figure 3.7.

In Chapter 4, I show that this does not add significant hardware costs to the final, synthesized, systems as: **(1)** the generated logic is all single bit boolean operations that are small compared to logic operating on the data that the architecture processes, and **(2)** can be evaluated in parallel. Furthermore, **(3)** when experimenting with this design flow we in the MASC lab found that much of the generated logic is optimized away by later passes and **(4)** much of the logic that is not removed stands in for logic engineers would normally implement by hand, such as “ready” signals that are often used to manage the back pressure problems in systems when their designers choose to do so on an ad-hoc basis.

We in the MASC Lab have also found that using Liam makes implementing fluid pipelined architecture much easier, as it requires less lines of code and it’s easier to reason about the behavior of the system when debugging it. For these reasons, the Liam paradigm is central to the Pyrope programming language and our Fluid Pipeline design toolchain.

Chapter 4

Pipeline Transformations

The monkey is repulsive, he's too much like us!

Quintus Ennius, *Saturae*

This chapter analyzes fluid pipeline transformations in more detail, both the specifics of implementing them, and how they can be leveraged to aid in the digital architecture design process.

Section 4.1 discusses how we define transformations in the context of the Pyrope language and compiler, building on prior work discussed in Section 2.1.2. Section 4.2 discusses how we implement them as part of the compilation flow.

The remaining sections in this chapter present ways in which fluid pipeline transformations can be leveraged for simulation, synthesis, and verification, and evaluate each method. The evaluation framework is discussed in Section 4.3.

Section 4.4 focuses on synthesis. This section discusses a fluid pipeline based solution toward a common problem in digital architecture development: the challenge of sharing code between projects with different timing targets. The problem comes from the fact that in order to achieve a fast clock speed, the logic of the architecture must be divided between many registers, however if a slow clock speed is tolerable, the system can save energy by removing registers. Section 4.4 shows how pipeline transformations, specifically merging, can be used to remove registers between stages, transforming four and five stage RISC-V cores into two and three stage cores. Their synthesis results are compared (area/power, delay, CPI) to manually created RISC-V cores with similar features.

Section 4.5 looks at using pipeline transformations to speed up simulations. Using successive *merge* operations to collapse a pipeline, no matter how long, into a single stage creates a system that is functionally similar to an emulator. Section 4.5 evaluates the speed up gained by doing this and compares its overall performance to Spike [14], the official RISC-V reference emulator programmed in C.

Section 4.5 also looks at using the collapse technique as part of a sampling schema. Sampling is a simulation technique that uses a fast emulator and a slow, cycle accurate simulator to allow the simulator to cover more ground while sacrificing as little accuracy as possible [74]. Normally, this requires each implementation of the architecture be coded by hand. With fluid pipelines, the same RTL implementation could serve both roles.

Finally, Section 4.6 discusses the use of pipeline transformations in verification. In this verification model, the developer would have two architectures with the same functionally, but different features, such as a different number of stages, different forwarding paths, or different branch predictors. There are tools called SAT solvers that can verify if two logic networks have equivalent functionality, but generally cannot be used in this case because the different features of those architectures result in different timing properties. However, both architectures should have identical timing properties and functionality if collapsed into a single stage, barring the presence of bugs.

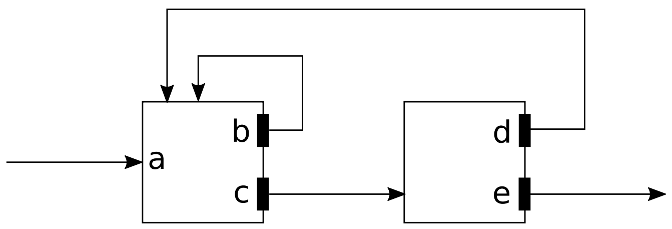
4.1 Pyrope Pipeline Transformations

Prior work on fluid pipeline transformations has defined the Recycling and Retiming transformations, discussed in Section 2.1.2. These allow the user to add/remove fluid registers and move fluid registers, changing the timing properties of the system without changing behavior.

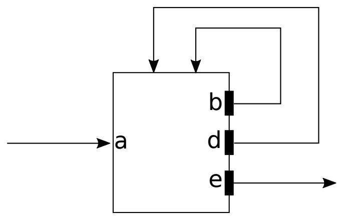
A major motivation of the Pyrope project was to integrate these transformations with the digital architecture design toolchain. Doing so requires creating an interface that the developer could use to apply transformations as part of the compilation flow. There is still work to do in this area, since so far, there has been only one transformation, *merge*, integrated into the Pyrope compiler. Even so, we have found a variety of applications that work by leveraging just this one operation. The merge operation is defined in detail in this section.

The Pyrope compiler defines the *merge* pipeline transformation as an operation between two instantiated stages. Figure 4.1 shows what this looks like in circuitry.

It is important to note that the *merge* operation is commutative in terms of behavior, but not commutative in terms of timing. As shown in Figure 4.1, only the connections in one



(a) pre-merge



(b) post-merge

Figure 4.1: Merging two stages together results in the "forward" connections between them going away while the "back" connections between them are maintained.

direction can be removed (referred to as "forward connections") while the connections in the other direction (the "back connections") must be maintained.

This leads to an important rule with the *merge* operation: Removing fluid registers whose data-path travels in the same direction of the pipeline has the effect of decreasing the pipeline length by one cycle. Removing registers whose data-path goes in the opposite direction implicitly means maintaining the forward-path registers, and thus does not decrease the pipeline length.

When integrating with Pyrope source code, this means that the *merge* operation is defined between two *instantiated* stages and the connections between them. The same code block could be instantiated in many different places, connected to its neighbors in different ways, parameterized differently, and be merged with other instances in different orders. This adds additional challenges from the tool-design standpoint as it makes it difficult for the developer to reason about the effects of pipeline transformations by just looking at the code.

```
core = merge(F, D)
core = merge(core, E)
core = merge(core, W)
```

Listing 4.1: Listing of transformations to collapse a 4-stage pipeline with the stages: F, D, E, W, into a single stage named *core*.

Pipeline transformations are specified as a series of operations. As an example, Listing 4.1 shows the series of merge operations which could be used to collapse a 4-stage pipeline with the stage-instances: F (fetch), D (decode), E (execute), and W (write back), into a single stage-instances named *core*. Transformations such as these are listed in a configuration file, that can be referenced by name using the Pyrope compiler's *repipe* command line argument. The compiler-level implementation details are discussed in the Section 4.2.

4.2 Implementation

As discussed in more detail in Chapter 5, the Pyrope compiler's internal data structures are designed to minimize duplication for faster performance. A Pyrope architecture is represented internally in the compiler with the following information:

- *Block Table*: A name-to-block mapping of blocks containing all logic in the architecture.

- *Instance Table*: An instance-name to block-name mapping of all instances in the system and which block they instantiate.
- *Connections*: A list of connections, that are either a connection between stage-instances or a connection between a stage-instance port and a global IO port.

Applying a *merge* operation, or any kind of pipeline transformation defined in the future, requires updating the internal representation to reflect the new pipeline, and rerunning the type resolution and constant propagation passes to fully optimize the changed topology.

4.3 Evaluation Setup

In our evaluation of this method, we use several 4, 5, and 6-stage RISC-V cores, listed in Table 4.1. Each has an instruction and data memory and can be compiled into either 32 or 64 bits, supporting the RV32i or RV64i ISA, respectively. The four stage RISC-V Fluid cores have the topology: fetch, decode, execute, and write-back. The five stage RISC-V Fluid core breaks the execute stage into two stages, and the six stage core breaks the execute and the write back stage into two stages. Each RISC-V core includes instruction and data SRAMs.

We compare these against several open-source RISC-V cores freely available. VS-cale [19] is a 32-bit, 3-stage CPU. PICORV32 [9] is a 4-stage, also 32-bit. RI5CY [13] is a 4-stage 32-bit core that has additional features such as a prefetch buffer. Zero-riscy [20] is a smaller 2-stage version of RI5CY.

Table 4.1: Cores used in the evaluation.

Name	Fluid	Main Characteristics
c4	Yes	4-stage
c4+fwd	Yes	4-stage with forwarding
c5	Yes	5-stage
c5+fwd	Yes	5-stage with forwarding
c6+fwd	Yes	6-stage with forwarding
Zero-riscy	No	2-stage [20]
VScale	No	3-stage [19]
PICORV32	No	4-stage [9]
RI5CY	No	4-stage [13]

The automatically Fluid transformed cores keep the same original name and add the generated number of pipeline stages. E.g: c4+fwd 2-stage means that the original c4+fwd Fluid core was automatically transformed to become a 2 stage pipeline core.

For synthesis, we make sure that all the SRAMs associated for data and instruction are not included in the synthesis results by excluding those SRAM blocks from the synthesis itself. When possible we also select configuration parameters to match the Fluid RISC-V cores. For all the cores, we use a commercial synthesis flow that has topological information during synthesis. For each core, we select a target frequency 5% under the maximum achievable frequency for that given core.

All the simulation and verification is done in the same machine for benchmarking reasons. We compile the system on an Arch Linux (October 2017) with GCC 7.2 and the -O2 optimization flag. These are run on an Intel(R) Xeon(R) CPU E3-1275 v3 @ 3.50GHz, with 16 GB RAM. For formal verification, we use YOSYS [73] version 0.7+312 as the SAT solver interface compiled with Clang 5.0.

4.4 Synthesis

This section shows that fluid pipelines can be used to automatically generate efficient cores with less pipeline stages than present in code, and to show that Fluid cores are competitive against existing RISC-V cores.

We do a competitive analysis between all the Fluid cores and several non-Fluid RISC-V cores. For the Fluid RISC-V cores and the available Non-Fluid RISC-V cores, we perform synthesis with a commercial tool that has a topographical aware synthesis, and report the time and area for each core. Figure 4.2 shows the resulting plot with the y-axes showing the normalized area against c5+fwd, and the x-axes showing the delay in nanoseconds.

One observation is that Fluid RISC-V cores have approximately similar delay-area trade-offs to Non-Fluid RISC-V cores. This is important because it shows that Fluid does not introduce significant additional design overheads.

This may seem counter-intuitive since Fluid Pipelining means replacing non-fluid registers with more expensive fluid registers. The effect of fluid registers on area is greatly minimized because, for one reason, when a Fluid Pipeline is compiled to Verilog, with application

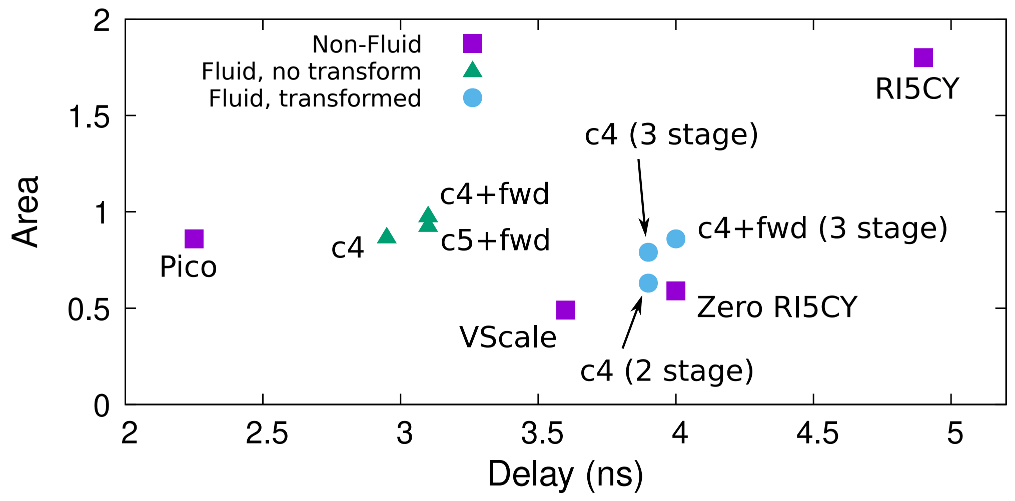


Figure 4.2: Fluid cores are competitive against manually generated Non-Fluid RISC-V cores.

logic that is provably more deterministic, especially with regards to back pressure, synthesis tools can simplify the logic significantly.

Another important reason is that must of the logic generated by the Pyrope compiler stands in for logic which would have to implement manually anyways. Pyrope infers valid checks for inputs to stages like the register file, and execute block; when developing processors with Verilog, the developer adds this logic manually.

Another important observation is that automatically transformed Fluid cores (c4 2-stage, c4+fwd 3-stage…) consistently save area at the cost of pipeline frequency. This is the expected result, but even more interesting is that the generated/transformed cores are consistent with non-Fluid cores such as VScale (3-stage) and Zero-riscy (2-stage) RISC-V cores.

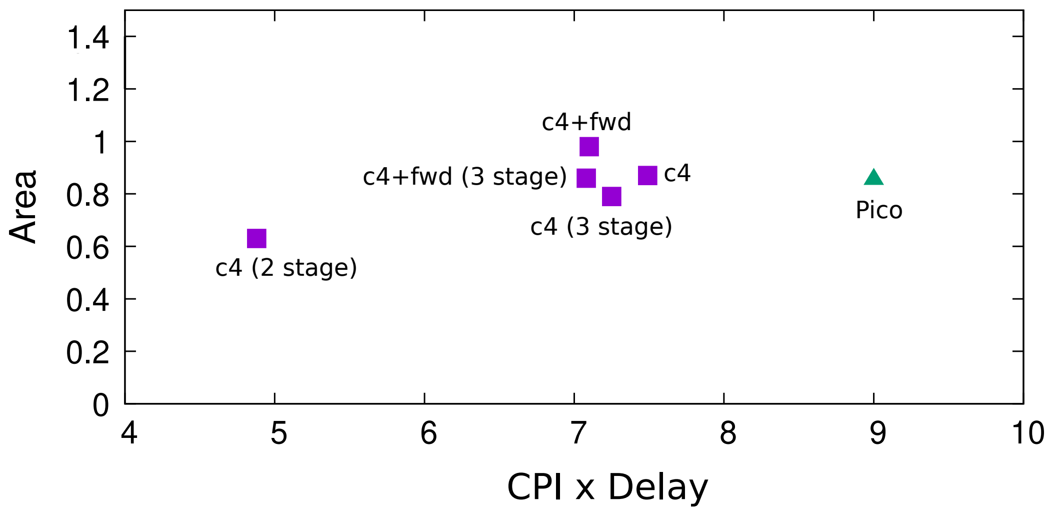


Figure 4.3: The Pico CPU has a CPI of 4, giving it a lower overall performance than the Cliffs, despite its favorable area/delay stats. This figure also shows that the performance fluctuations from merging stages is not constant.

Despite the fact the PICROV32 and VScale are on the Pareto frontier in Figure 4.2, that figure does not take into account CPI. Figure 4.3 does, mapping performance against area for the Cliffs and PICROV32.

An interesting observation is that one of the non Fluid RISC-V core (RI5CY) seems to have a significant area and frequency overhead. The area overhead is due to the extra prefetcher available in the model and a multiplier unit that does not exist in the other cores. These two blocks account for approximately 25% of the RI5CY area. The Zero-RISCY is a 2 stage pipeline version of the RI5CY; this more area optimized core is very close to the Fluid transformed c4 2-stage core.

Our RI5CY vs Zero-RISCY area synthesis results are consistent with PULP [10] published results were the Zero-RISCY has approximately 30% of the RI5CY area. In our case, Zero-RISCY has higher frequency, but in the previously published results, the target frequency was a conservative 100MHz for both cores. In this work, we always target the maximum achievable frequency and then decrease the target frequency by 5% to avoid complex timing overheads. The same methodology is applied to all the cores.

One last interesting observation is that the delay of the c4 does not change going from three to two stages. This means that the merge from three to two did not change the critical path. Considering that Retiming (Figure 2.5b) allows developers to change the timing of the pipeline automatically, the ability to merge and re-balance may prove a useful addition to this tool.

Besides comparing against RISC-V cores, we also compare a larger 6-stage Fluid RISC-V core (c6+fwd) and its automatically generated shorter pipeline RISC-V cores against ARM Cortex-M series.

To understand the automatically transformed results, we compare against the ARM cores. Figure 4.4 shows the c6+fwd and the automatically transformed 1-stage and 3-stage performance normalized against the ARM Cortex-M series. The Cortex M7 outperforms the c6+fwd in 6-stage form, likely because its pipeline is simply more balanced. Nevertheless, as we collapse the designs the imbalance is not as important, and the performance change between the collapsed RISC-V and ARM core is lower.

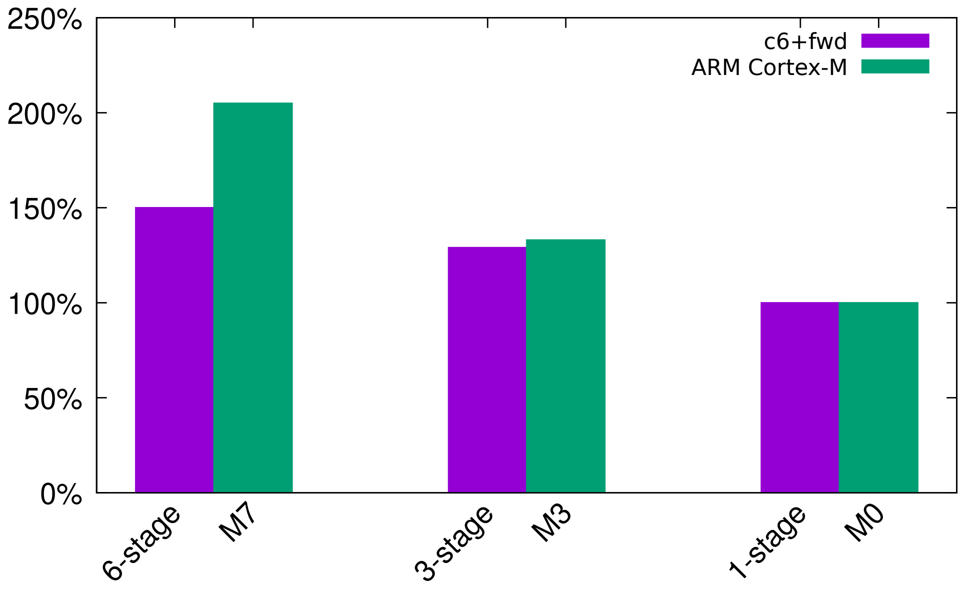


Figure 4.4: This graph normalizes the single core performance of AMD's Cortex series to our c6+fwd core in 1-stage form. It shows that the performance deltas between automatically transformed cores can be similar to well-tuned cores implemented manually.

4.5 Simulation

For architecture simulations, the *merge* pipeline transformation has the potential to provide speedups from two directions. First, merging stages together removes the infrastructure between them, simplifying the simulation. Second, doing so removes pipeline bubbles, reducing the cycles-per-instruction (CPI) rate, so the simulation can cover more ground quickly. The cost of using this optimization is that cycle accuracy is lost.

Figure 4.5 summarizes the overall speed in MIPS for different ways to perform simulation for a c4+fwd Fluid RISC-V core. While the unmodified c4+fwd core runs close to 2

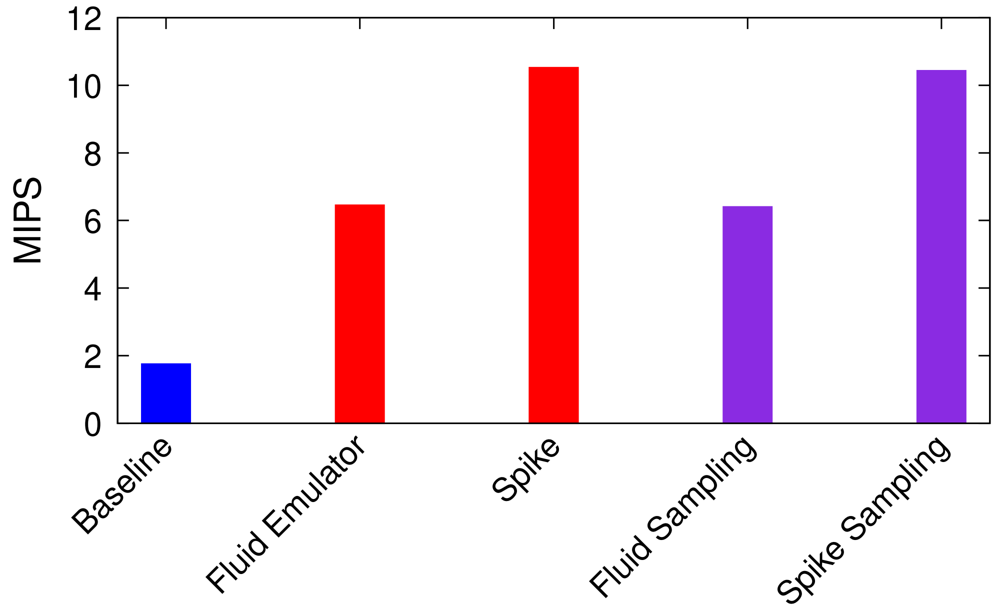


Figure 4.5: Collapsing provides roughly a 3.7x speedup over the default state-of-the-art baseline simulation. It is also less than 50% slower than Spike, a C emulator.

MIPS (baseline), the fully collapsed c4+fwd core runs over 6 MIPS. We observe a 3.7x speedup due to Fluid Emulation.

We also compare our system against Spike [14], a C++ emulator provided by Berkeley as the RISC-V reference model.

As expected, Spike is faster than Fluid Emulation but the difference is less than 2x. Fluid Emulation is 40% slower than Spike. Though collapsing the pipeline into a single stage and compiling it in C creates something that is functionally similar to a native emulator: the emulator is still compiled from unmodified RTL code, as opposed to Spike which was written directly in C and tuned for speed. This is also using an early version of the Pyrope compiler, which has not gone through generations of performance tuning as the Spike has.

For these results, we ran Spike doing cache warmup. As seen with other emulators [28] doing warmup is around 10x slower than just doing fast emulation. The same ratio happened with Spike, it can run 10x faster when cache warmup is disabled. It is important to remember that Fluid Emulation does warmup for all the structures.

Once we have a fast emulation with warmup, we can do statistical sampling. This is also shown in Figure 4.5. Fluid Sampling combines the Fluid Emulator with a SMARTS [74] sampling technique. This means that we can achieve detailed performance and power estimations with over 6 MIPS. When CPI is considered, this means that we achieve close to 10MHz simulation speed with a software only solution without having to build any code besides the c4+fwd core. If we had combined the results with Spike instead of Fast Emulation, we could achieve a 60% speedup, but it would require significant code changes to transfer state between the emulator and detailed core every time that we perform a statistical sample. With current digital design techniques, the logic for transferring state between the cycle-accurate simulator and emulator is generally difficult to maintain because it is an interface between two different types of systems. With Fluid Emulation this can be done much more easily because both implementations of the system are built with the same data structures and external interface.

We also compare against Strober [53]. Strober is an FPGA-based sampling scheme where the design under test runs in an FPGA which has been instrumented to gather statistics. An emulator only without instrumentation could run at around 50MHz, but the Strober work shows under 4MIPS speed because of CPI and lower lower FPGA frequency when the code is instrumented. This is even assuming a very large simulation period to ignore the costly FPGA

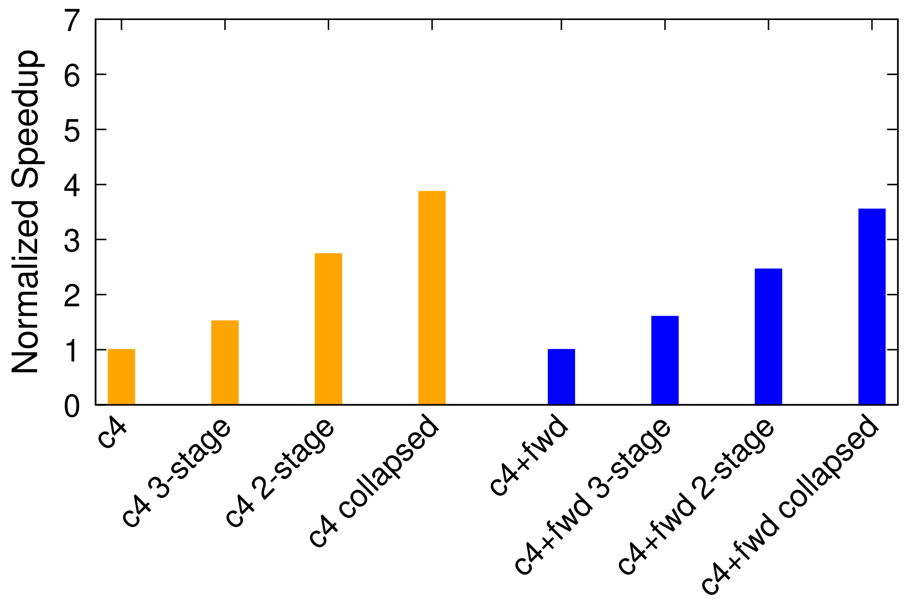


Figure 4.6: Collapsing stages together improves the overall simulation speed, with compounding benefits as more transformations are done.

compilation process. This means that the 10MHz speed for Fluid Emulation with SMARTS is competitive against FPGA setups.

Overall, the *merge* pipeline transformation can be used to significantly improve the performance of an RTL simulation, at the cost of cycle accuracy. Unlike current digital design techniques, this can allow developers to create an emulator, and even a sampling-based simulation, from a single codebase.

4.5.0.1 Simulation Insights

This section analyzes in more detail the speedup gained by merging pipeline stages.

Collapsing pipeline stages together improves simulation speed, with the effect compounding on each transformation, as shown in Figure 4.6. The speedup comes primarily from two sources. First, removing pipeline infrastructure removes work from the simulation, decreasing the time required to evaluate each cycle. The speedup is shown in Figure 4.7a. Second, removing pipeline stages decreases the cycle-per-instruction (CPI) ratio, shown in Figure 4.7b, which allows the simulation to cover more ground in less cycles.

Figure 4.7a shows that the increase in MHz speed is relatively constant regardless of which stages are collapsed, while Figure 4.7b shows more variability. This shows that design decisions made by developers can have an impact on the system, post-transformation.

One thing not shown is that there are different ways to get from, say, a 4-stage core to a 3-stage. For example, to create a c4 3-stage design, we could collapse the decode and execute stage, or the execute and the writeback stage. We evaluated both options, and the simulation speed and/or CPI was fairly similar. In the evaluation, the 3-stage corresponds to the case when the decode is combined with execute.

As mentioned in Chapter 3, the Pyrope compiler can compile architectures to either Verilog for synthesis/simulation or C++ for simulation. The C++ target has additional optimizations, further discussed in Chapter 5, to provide incremental compilation and live programming support. These benchmarks were taken before those optimizations were added, thus only the Verilog output target was benchmarked. The optimized, C++ simulation target is benchmarked in Section 5.6.

4.6 Verification

Fluid Formal Verification means collapsing two fluid architectures, generally one which is the unit-under-test (UUT) and another which is the reference model, into a single stage, then using a SAT solver to prove equivalence. Unlike most formal verification systems, we do not implement a single assertion or model checking. Fluid formal verification relies on a reference model.

When a performance feature like forwarding is added, or when the number of pipeline stages are changed, we can apply formal verification against the two versions because the result should not change. If there is a mismatch, our performance enhancement has introduced a bug. This is useful when the core is still not finished and a formal verification against a reference model is still known to fail. Such checks are also transitive, which we believe is another beneficial feature which could be leveraged.

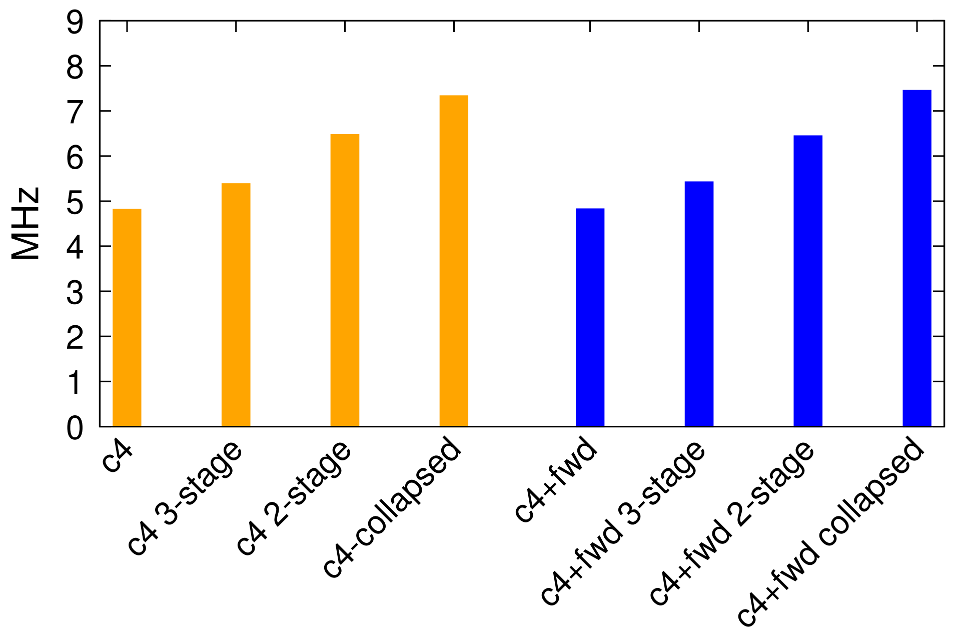
The Fluid RISC-V cores are simple in-order designs where relative completion order is not important. This means that picking any collapse order yields the same equivalent design. For more complex cores, to do a formal verification it would be necessary to check equivalence order for all the possible combinations. For example, if there were a floating point unit with different pipelines for addition and division, the relative completion speed between the two of them should not affect the results. This is captured when the collapse happens. If, say, the addition is collapsed before, it is equivalent to saying that the division is always slower. If the division is collapsed before, it is the same as saying that addition is always slower. As stated in Section 2.1.3, fluid pipelines are built on the idea that the ordering of packets in the system does not affect behavior, the results should be the same regardless of how the collapse is done. It is important to note that Fluid Formal Verification is implicitly making that assumption, by checking only one collapse ordering against the reference model.

We leave for future work how to formally verify designs where different collapses generate difference valid results. An example is a multiprocessor where getting one core faster than other can yield different execution orders but all the orders are valid given a consistency model.

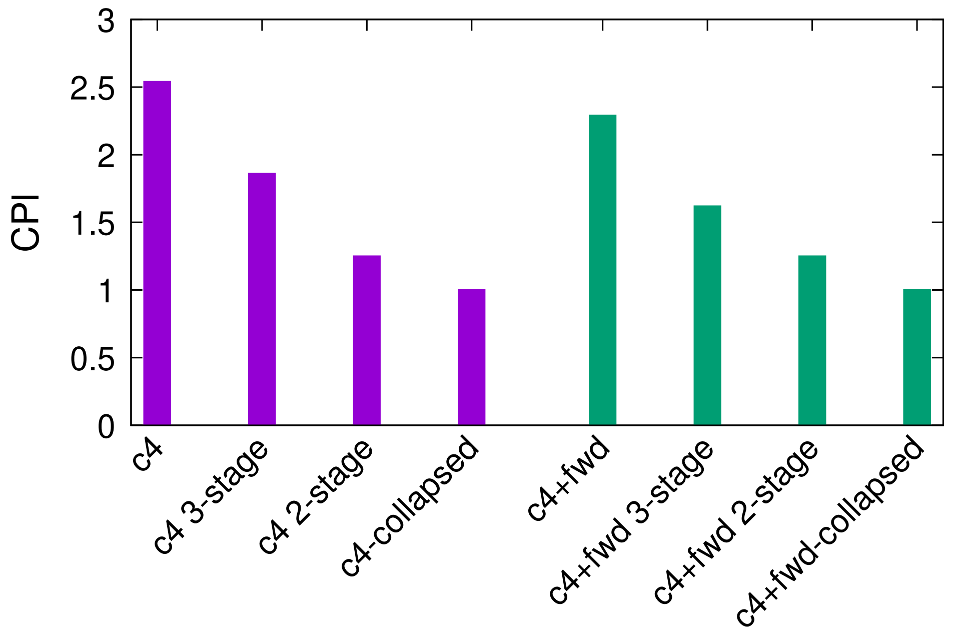
4.6.1 A Note on Fluid Induced Bugs

Although fluid pipelines should behave independent of latency or ordering, developers make mistakes, and mistakes with regards to fluid pipeline assumptions can result in difficult to find bugs, since these bugs may only manifest themselves if the pipeline is transformed in certain ways.

Fluid Format Verification can be a useful method to find these bugs since it is based on a provably direct equivalence check. Another informal method which can be used to find such bugs is the have the fluid flops introduce random delays during the simulation. I believe there is significant future work in exploring these verification techniques.



(a) Removing infrastructure decreases he time required to evaluate each simulation cycle.



(b) Merging stages decreases CPI, also yielding performance improvements.

Figure 4.7: Decreasing simulation cycle time and CPI are the two primary sources of speedup for a collapsed pipeline.

Chapter 5

Live Programming

They praise good books, but read bad ones.

Marcus Valerius Martialis, *Epigrammata*

The last chapter of this document focuses on LiveSim, a live development environment for hardware description languages. LiveSim is language independent, though it was built with the Pyrope language and compiler in mind.

The goal of this section, and of this part of the Pyrope project is to provide tools to better aid programmers in the debugging/verification process. We do this by addressing a major source of inefficiency in the debugging/verification flow, that will be referred to as the *edit-run-debug* (ERD) loop: the long latency required between making a change of the source code and seeing that change reflected in the simulation. For modern digital architectures, it is not uncommon to see compile times of over an hour. Furthermore, as architectures have gotten more complex, simulations have had to get longer to maintain decent coverage, resulting in hours-long simulations becoming common as well.

The long latencies of ERD loops in modern digital architecture flows is a major hindrance to productivity, as it forces developers to use their simulation tools sparingly. Debugging is inherently an investigative process where hypotheses are created and checked. Studies [22] show that debugging accounts for over 40% of the verification time, and that its importance has been increasing in the last few years. Being able to check a hypothesis accelerates the debugging process.

There have been many attempts to accelerate simulation [30, 44, 67]. However, the underlying issues remain: 1) designs, in particular for large multi-core systems, are complex, and 2) to guarantee code coverage, simulations need to be long to account for different scenarios. This leads to long compilation times, usually from a hardware description language (HDL) like Verilog or VHDL, and even longer simulation times. There are other, newer, though lesser-used HDLs like Chisel [29] and PyMTL [59], that aim to improve design productivity. Unfortunately, these tend to also increase the compile time by adding another abstraction layer. Chisel, for example, compiles Chisel to Verilog using the Java Virtual Machine (JVM), that must then be passed to another tool.

Live programming stands in stark contrast to traditional hardware design flows. With live programming, developers write the program as it is running and see their code change immediately, or at least quickly, reflected in the system's behavior. Prior work has shown that faster feedback leads to a faster and less frustrating development experience [31, 41, 57], to many developers this is intuitively true as well. Toward that end there exist many software development tools, like Microsoft Visual Studio [2] and Netbeans [5], that provide live feedback of changes to the user.

This is mainly possible through a combination of incremental compilation techniques. Hot binary reload, also referred to as swap or simply Hot Reload, means swapping the binary after a code change, without stopping execution. This allows the current state of the program to be maintained across different versions. Checkpointing is another technique used by live compilation flows where a state is saved throughout execution. A checkpoint can then be loaded, and execution continues from that point, avoiding the need to execute the code from the beginning. This is especially useful for very long input sets, which is often the case in architectural simulations as well.

For hardware designs, their increasing size and complexity presents additional challenges for implementing a live development environment. However, incremental techniques for hardware designs have been proposed [66], with a focus on synthesis. Moreover, hardware simulations are notoriously slow. Typical industrial flows require tens of minutes just to do a minimal code change; it is not uncommon for a simulation to take hours to reach a failure point. The ability to hot reload and/or checkpoint rather than restart would have a huge impact on feedback times and thus productivity.

There are subtle challenges of creating live simulation for hardware, such as dealing with adding or removing states to and from the simulation, dealing with corruption of state due to faulty code patches, and deciding if the checkpoint is consistent or if we should have used a earlier checkpoint. These all need to be integrated into a solution, and have already been addressed in other live programming environments.

In this section we propose LiveSim, a framework for Live Simulation of hardware designs. LiveSim leverages incremental compilation, hot reload, checkpointing, and a few extra techniques to provide live feedback for hardware. When designing LiveSim, we focused on shortening what we refer to as the edit-run-debug (ERD) loop. This refers to the latency between when an engineer makes a change in the code, and when the updated results of the simulation are available. Hardware design flows have long latencies in their ERD loops, which harms productivity.

LiveSim can significantly speed up the ERD loop, as shown in Figure 5.1. The improved speed comes from two sources. First, incremental compilation: LiveSim can compile only the parts of the architecture that have changed and patch them into the executable, rather than rebuilding the entire thing. Second, LiveSim leverages simulation checkpoints to prevent executing the whole simulation from the beginning. In our evaluation, we show that LiveSim is able to maintain a latency of under 2 seconds in its ERD loop, even for an architecture containing 256 RISC-V processing cores.

This paper sets a feedback goal of under two seconds between code change and updated result, even for large-scale architectures. Two seconds is the recommended [41] value to have a *responsive user* interaction. We leave as potential future work a user study comparing productivity impact of various response times.

LiveSim includes the capacity to: **(1)** hot reload combinational logic and state, and **(2)** using the lightweight checkpointer, quickly verify in parallel that the checkpoint's initial state is consistent. Item (2) describes a simple algorithm leveraged by LiveSim's simulator, allowing it to provide a quick estimate on the updated state of the system after a hot patch, while continuing

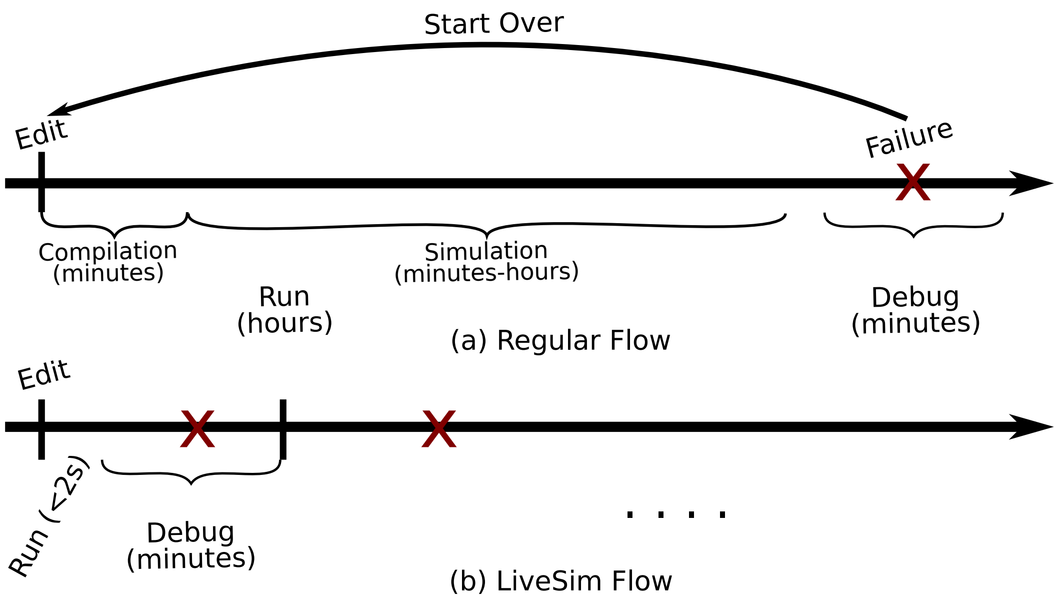


Figure 5.1: The proposed workflow for LiveSim is to allow iterations within a few seconds. This contrasts with the traditional flow where after a change, there is a lot of latency in each iteration. With the much faster ERD loop, enabled by LiveSim, productivity is expected to improve. (Picture credit: R. Possignolo *publication pending*

to refine that estimate on the backend, hiding as much of the computation as possible from the user.

To support the goal of updating simulation results within two seconds of a code change, LiveSim implements a fast compilation technique that includes **(3)** incremental HDL parsing and hierarchical partial compilation. LiveSim uses incremental parsing and compilation to compile changes in the architecture into hot loadable shared libraries, for later patching into the simulation.

Finally, **(4)** we develop and prototype a language-neutral Hot Reload for hardware. Users of hardware simulations would greatly benefit from being able to hot load source code changes rather than having to recompile and rerun the entire program. This paper presents the first hot reload architecture for digital hardware simulation. We show that the LiveSim infrastructure is not only able to perform Hot Reload in under 2 seconds, but it is also more scalable than currently available simulators.

LiveSim is faster in raw simulation speed than Verilator [17], the state-of-the-art open source simulator for hardware. For instance, when evaluating a 16 node large Partitioned Global Address Space (PGAS) RISC-V multicore, Verilator has a global speed of 51.4KHz or 704 total clock cycles ($704 = 51.4 * 16$). LiveSim, with or without checkpoint overheads, has a global speed of 93 KHz, or 1488 global clock cycles. Verilator is slightly faster for trivial designs, but as the design complexity increases, the generated code footprint does as well, and it suffers from significant instruction cache misses in the host machine. LiveSim does not have such scalability issues. In fact, with a 256 core PGAS, LiveSim can be Hot Reloaded in under 2 seconds.

The remainder of this chapter is organized as follows.

Section 5.1 disucsses related work specific to LiveSim. Section 5.2 discusses the Pyrope compiler, focusing on the internal data structures which are designed to improve performance, minimize duplication, and provide support incremental compilation and simulation. Section 5.3 then adds some commentary on the challenges posed by Pyrope's type system, that is still under development.

Section 5.4 moves on to LiveSim, beginning with the intended functionality and imagined use cases (Section 5.4.1), followed by the additional internal architecture (Section 5.4.2), and finally the incremental compilation and update mechanics (Sections 5.4.3 and 5.4.4).

The final part of this chapter evaluates the viability of LiveSim to deliver a live development experience. Section 5.5 discusses the setup for this evaluation, with the results provided in Section 5.6.

Section 5.7 adds some concluding remarks about LiveSim, and the state of hardware development tools in general.

5.1 Related Work

5.1.1 Live Programming and Hot Reloading

Live programming, also referred to as interactive programming, is the process of developing a system as it is being run. There are a variety of development environments for software programming which provide this type of functionality, with some variation. Development environments like Visual Studio [2] and NetBeans [5] allow developers to edit the source code of a program as the program is being executed for debugging. After a code change, the system is recompiled and the execution continues without the need to restart the debugger. Colt [4] allowed live coding for ActionScript [60]. There are also musical and graphics related live coding tools which have been used for performance art [47, 72].

LiveSim primarily draws inspiration from debugger-based live programming environments like Visual Studio or NetBeans, as the simulation of the system can be seen as analogous to a debugger.

In hardware, there are no approaches for live programming. However, some existing approaches could be adapted to provide a live environment. Jupyter Notebook [56] is an open-source web application which allows users to create interactive documents which can include code. Chisel [29], a hardware description language invented at UC Berkeley, allows for integration in Jupyter as an educational aid. This has similar motivations to our project, the main goal being to manipulate a system and get fast feedback, which aids the user in understanding it. Jupyter, and Chisel's integration to it, are not optimized for large designs, however, severely limiting its application. Updating even trivial designs that involve only a single stage can take several seconds.

Hot reloading is the ability to modify the behavior of a program, by loading and/or unloading code, without stopping it. Live programming environments often make use of hot reloading, but hot reloading has other applications as well.

React Native is a Javascript library designed for implementing native applications with Facebook's React library, which as of 2017 supports hot reloading [7]. The primary motivation behind implementing this feature was to improve the development experience, as it would allow developers to make changes to the code without losing the state of the app they are debugging. There also exist hot loading libraries for Java [27], and various other languages.

We are not aware of any hot reloading libraries for simulating Verilog or any other synthesizable HDL.

5.1.2 Checkpointing and Parallel Replay

LiveSim leverages checkpoints in order to provide a live development experience. Specifically, LiveSim: **(1)** takes checkpoints at regular intervals, **(2)** can modify them to reflect changes the user makes to the source code, and when the user makes a change to the architecture source code, Livesim can **(3)** replay them in parallel to quickly find the earliest point in the testbench execution history where the behavior of the new system diverges. We are aware of no hardware development environment which incorporates these capabilities, however there are other projects which have implemented similar functionality.

There are a variety of projects [25, 54, 55, 68] which use multi-tiered simulations, a fast simulator to create checkpoints and a slow simulator which can evaluate them in parallel, to speed up HDL simulation. One key difference between these projects and LiveSim is that LiveSim uses only one simulator to generate the checkpoints. The validation step simply checks whether a pair of checkpoints match, which can be done with a direct memory comparison.

A similar technique has been proposed for approximate computing [75], where a fast approximation of the application runs and takes checkpoints, while detailed implementations validate them in parallel. This does have similarities to item (3), though like the fast HDL simulators listed above, would have to do considerably more work to validate the checkpoint deltas.

5.2 Compiler Support for LiveSim

The internal architecture of the Pyrope compiler is designed with two goals in mind: performance and support for incremental compilation/synthesis/simulation.

Pyrope architectures are represented as blocks, which can be either a stage or a pipe. Besides being designed as a platform for fluid pipeline research and development, Pyrope was also intended to plug into a live development environment, which required it to be fast, and work both as a stand-alone compilation flow and in an SaaS format.

For this reason, the Pyrope compiler is designed from the ground up to support incremental compilation and *hot reloading* of simulations. Hot reloading means being able to update the simulation executable with new code without having to restart it.

To accomplish this, Pyrope builds the architecture into the data structure shown in Figure 5.1. The architecture is divided into the following components: blocks, which contain all the logic of the architecture; and instance table, which maps an instance to the block it is instantiated from; and a list of connections between instances, or between an instance and a pipeline IO port.

There is an important subtle distinction in that a "block" data structure a Pyrope does not necessarily map one-to-one with a block in source code. The same block in source could be instantiated in some places with fluid behavior and in some places without. This would require each instance to be mapped to different "block" data structures. The Pyrope language is also still in development and we have been experimenting with programming constructs like polymorphism and global type resolution which could further differentiate instances which come from the same source-block. Section 5.3 discusses challenges of implementing this from the compiler perspective, and leaves the discussion on Pyrope language semantics to future work.

Pyrope's internal representation differs significantly from those of other HDL simulation and synthesis tools. Verilator [18], a widely used open source Verilog simulator, compiles the entire architecture into a single, massive, C++ function. VCS [15] builds and simulates from a global netlist, which is gate level representation of the architecture. In Chapter 5, we show

Table 5.1: Pyrope's data structure is designed to minimize the amount of information needed to represent the architecture.

instance-table	$map[inst \rightarrow block\text{-}name]$
inputs	$map[name \rightarrow inst.port]$
outputs	$map[name \rightarrow inst.port]$
connections	$list[\{src\text{-}inst.port, dst\text{-}inst.port\}]$
blocks	$map[name \rightarrow block]$

that Pyrope significantly outperforms Verilator when compiling meshes of RISC-V processors sized from 1×1 to 16×16. This is because Pyrope's internal representation grew only by the number of connections, while Verilator's grew by the total number of operations.

5.3 Challenges Posed by Pyrope's Type System

Although Pyrope's type system is still in development, one aspect of it which is set in stone is that the type system will support type inference. Likewise, the Pyrope compiler and LiveSim architecture were designed to take type inference into account.

As stated in Section 5.2, the Pyrope compiler internally represents the architecture as a mapping a unique block ID to each block, and a mapping of each instance in the architecture to the block it is created from, using the data structure shown in Table 5.1. This presents a challenge when dealing with code like in Listing 5.1. During the type resolution process, Pyrope needs to identify that the 32-bit and 64-bit instances of adder need to refer to different internal blocks.

```
a32,b32 as bits:32 input
o32 as bits:32 output
a64,b64 as bits:64 input
o64 as bits:64 output

adder32 as adder a:a32 b:b32 o:o32
adder64 as adder a:a64 b:b64 o:o64
```

Listing 5.1: Pyrope allows for global type resolution to enable writing generic blocks. This presents a challenge when implementing this with regards to Pyrope's internal representation, shown in Table 5.1.

Implementing the type resolution algorithm correctly requires us to consider how the Pyrope compiler's internal representation of the pipeline would affect the process. Like most type resolution systems, Pyrope's is dependency based. The compiler looks at both the operation a node represents and the type properties of its dependencies to determine what the type of that node should be. When implementing the internal API of the compiler, one thing we had to take into account is the semantic difference between asking for the dependencies of an input of an *instance* versus the input of a *block*. Asking for the dependency of an input of an instance should return only the output which drive that instance's input port. Asking for the input of a block should return all outputs which drive that port for each instance of the block.

The Pyrope compiler handles cases like Listing 5.1 by identifying cases where in dependencies of a *block's inputs* have irreconcilable types, but the individual *instances* of that block do not. The compiler then creates separates blocks for those instances to refer to.

5.4 LiveSim

The goal behind LiveSim is to improve the productivity of hardware developers by providing fast feedback as changes are made to the codebase. To accomplish this it leverages a backend design for incremental compilation, which we describe in detail in this section.

The framework can be thought of as two conceptual operation modes. The baseline mode, where compilation and simulation are run regularly for the full code base, and the live mode, that starts when there is a code change. In the baseline mode, checkpoints are regularly created to be used during the live mode. An overview of the live mode is depicted in Figure 5.2. The steps in the live mode are:

- Incremental Compilation, triggered after a code change
- Hot reload of recompiled objects
- Checkpoint selection and reload
- Execution from the checkpoint to the current cycle
- Checkpoint consistency check (done in parallel in the background)

In this section, we first discuss how we envision that LiveSim will be used, then we discuss the high-level architecture and main components of the simulator. We follow by giving more details on the compilation infra-structure and reloading both of the object and of the simulator state. We also discuss how the consistency of the checkpoints is verified after a change.

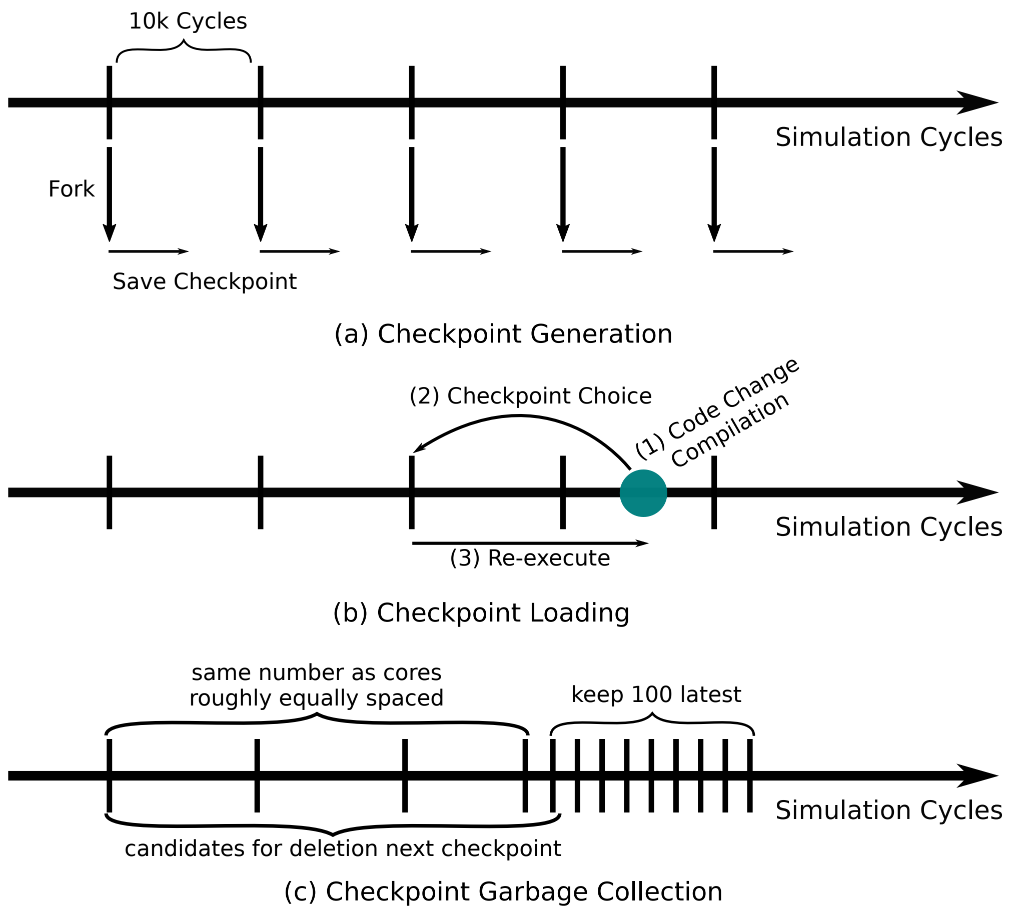

10k Cycles
Simulation Cycles
Fork
Save Checkpoint
(a) Checkpoint Generation
(2) Checkpoint Choice
(1) Code Change Compilation
Simulation Cycles
(3) Re-execute
(b) Checkpoint Loading
same number as cores roughly equally spaced
keep 100 latest
Simulation Cycles
candidates for deletion next checkpoint
(c) Checkpoint Garbage Collection

Figure 5.2: During the baseline execution, checkpoints are regularly created. Then, during the live mode, the code changes are recompiled, reloaded, a checkpoint is selected, loaded and the execution continues until the cycle before the code change was inserted. Checkpoints are regularly deleted to avoid too much overhead. As LiveSim is still in the experimental stages, the numbers associated with memory usage and checkpoints are largely heuristic. (Picture credit: R. Possignolo *publication pending*)

5.4.1 Use Cases

Although LiveSim requires a sizable amount of infrastructure, we believe its benefits to digital hardware developers will far outweigh any inconvenience or learning curve. In this section, we discuss specific use cases which could be helped by leveraging LiveSim.

Debugging a single simulation: This is the primary use-case for LiveSim. The developer can benefit by using fast checkpoint reloading to identify an error, test multiple possible fixes, and continue debugging without having to fully rebuild and reload the simulation.

Debugging "what if" conditions: Fast reloading can also be leveraged to check "what if" situations, such as being about to test what would happen should a branch misprediction be detected on an arbitrary cycle in a simulation.

Regression System: Thus far we have viewed the session history as a linear list of individual checkpoints, but we are not restricted to this. A regression system could be built on top of LiveSim which could run a set of testbenches the system and report their result as a batch.

Allow simulations to skip initialization: Restarting a simulation is universally slow. As an example, starting the BOOM [38] core from reset is especially slow because the processor initializes a debug monitor. Some companies take great pains to modify their flows to skip some of the initialization work for their testbenches. With hot reload, parallel checkpoint history verification, and deterministic register transformations (discussed in Section 5.4.5), this behavior can come for free.

5.4.2 The LiveSim Architecture

Figure 5.3 shows the high-level block diagram of the LiveSim architecture. The user interacts with the system both by manipulating the source code of the active project, and by sending commands to the simulator. As the user manipulates the architecture source code, LiveSim generates change events which track where the changes were made. At regular intervals, and based on user activity, LiveSim will batch these change events and send them to

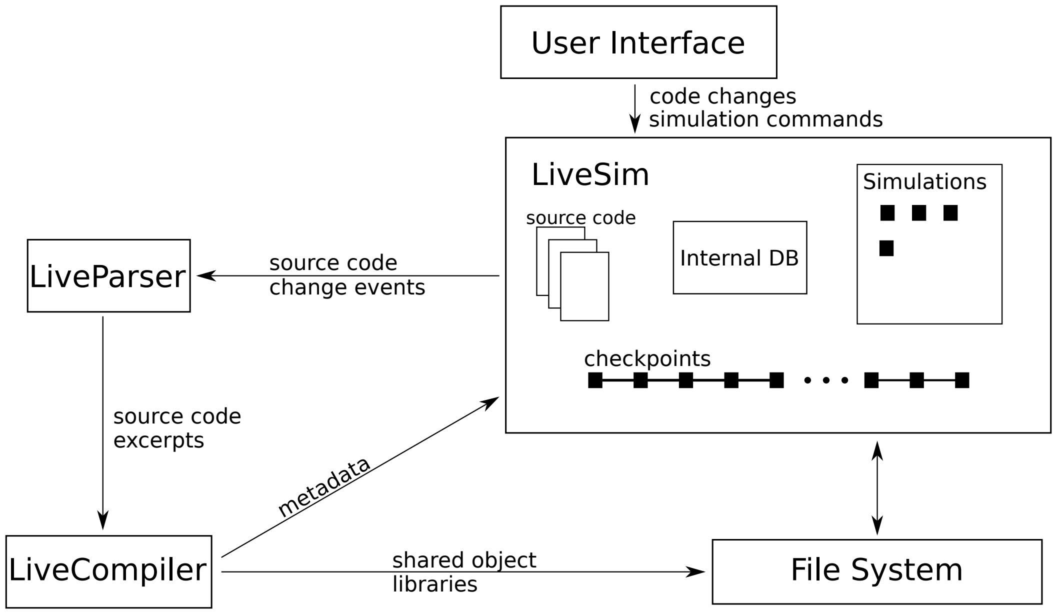


Figure 5.3: The LiveSim development environment.

LiveParser (Section 5.4.3). LiveParser determines if these represent changes in behavior, and which blocks or regions on the source were changed, and sends only those sections to LiveCompiler (Section 5.4.3), which will compile those blocks into hot-loadable shared object files. LiveCompiler will also update LiveSim with metadata which describes which parts of the simulation were updated. LiveSim can then retrieve those objects from the newly created object files.

Internally, LiveSim also maintains a session, tracking a history of which testbenches were run and for how many cycles, and takes regular checkpoints throughout. To prevent checkpoint creation from being in the simulation critical path, whenever a checkpoint is to be created, the process is first forked and then the child process creates the checkpoint and halts. Thus this happens with very minimal interference in the baseline process (see Figure 5.2). Should the simulation be patched, this history will be used to update the results more quickly (Section 5.4.4).

5.4.2.1 Internal Objects

The "live" part of the live programming environment refers to the fact that the system, in our case the simulation, is running as it is being developed. For this reason we don't view a simulation or testbench as a procedure which has an entry and exit point, but as an interaction between objects. At a high level LiveSim works with two types of objects, which are all hot loadable at runtime from shared libraries:

Stage: A block of logic. A stage has external IO, and could also have internal registers and memory depending on the source code. A stage can also create and connect to other stages.
Testbench: An operation which can be performed on a stage.

The Unit Under Test (UUT), or *pipeline*, object is a stage instantiated from a runtime-linkable shared object library. It is made up of that stage, which on creation may instantiate any number of additional stages from that, or other, shared object libraries. The testbench is also loaded from a shared object library, which contains procedures that can be run on a pipeline for any given number of cycles. Doing so changes the pipeline's internal state. Such changes are viewed by LiveSim as operations on the pipeline object, whose history is tracked and checkpointed as part of the simulation session. This allows those same operations to be applied again, should the pipeline object be updated due to a change in source code.

When the user makes changes to the source code, these are compiled into new shared object libraries. These new shared libraries can be hot loaded into the simulation and instantiate new stage or pipeline objects which can be swapped piecemeal into the executable. This allows the simulation to be quickly patched without needing to be fully recompiled, or even needing to fully copy the simulation state.

5.4.2.2 Internal Tables

To manage this internally, the simulator must be aware of the stage and testbench objects it is managing, and have a means to refer to and manipulate them. LiveSim maintains a path variable, and uses that to track all the object files in the directories it is aware of. At the beginning of a session, using a standard interface present in all of LiveSim's shared libraries, the session can retrieve data structures detailing the stages and testbenches present in those libraries. It maintains that information in an internal table, called the *Object Library Table*, shown in Table 5.3. It also keeps track of pipes, which are stages that are the top level of an active simulation.

The *Object Library Table* lists all of the objects that the current session of LiveSim is aware of. Each pipe, stage, or testbench object found is given an arbitrary name. This name is mapped to both its **source-path**: its location in source, and it's **object-path**: the path to the shared object file and the hook within that file which can instantiate this instance (in the case of a stage), or run the testbench (in the case of a testbench). Using the API described in Table 5.2, which is bound to a globally available LiveSim context, threads within the LiveSim environment can create and manipulate these objects.

Table 5.2: LiveSim commands. (Stage/tb)-handle variables refer to the handle column on Table 5.3. Stage-name variables refer to the corresponding columns in Tables 5.4 and 5.5.

Syntax	**Description**
ldLib *name*, *path*	Load shared library. Add entries to Table 5.3 for all object-paths found
instPipe *name*, *pipe-handle*	Instantiate a pipe and bind it to "name"
instStage *pipe-name*, *stage-name*, *stage-handle*	Instantiate a stage from *stage-handle*, bind it to *pipe-name* and *stage-name*
copyPipe *new-name*, *old-name*	Copy a pipeline, including its state
run *tb-handle*, *pipe-name*, *cycles*	Run a testbench on a pipe for a given number of cycles
chkp *pipe-name*, *path*	Take a checkpoint of *pipe-name* and save the state to *path*
ldch *pipe-name*, *path*	Load the state from a checkpoint into a pipeline
swapStage *pipe-name*, *stage-name*, *stage-handle*	Replace a stage in a given pipeline with a new instance

Table 5.3: The Object Library Table contains a listing of all the objects LiveSim could find in the shared object libraries on its path.

Handle	Type	Code-Path	Object-Path
pipe0	Pipe	/projects/src/toplevel.v#core	/livesim/objs/.../libc0.so#core
stage0	Stage	/projects/src/adder.v#adder	/livesim/objs/.../libc0.so#adder
stage1	Stage	/projects/src/sadder.v#sadder	/livesim/objs/.../libc0.so#sadder
tb0	Testbench	/projects/tests/tb1.cpp	/livesim/objs/.../libtb0.so#tb1

Table 5.4: The Pipeline table. Contains a name to pointer reference for each pipeline object active in the session.

Name	Handle	Pointer
p0	pipe0	0x...
p1	pipe0	0x...

Table 5.5: The Stage table. Contains a pointer reference for each stage of each pipeline.

Pipe Name	Stage Name	Handle	Pointer
p0	s0	stage0	0x...
p0	s1	stage1	0x...
p1	s1	stage1	0x...

Table 5.4 shows the *Pipeline Table*, which lists the pipeline objects instantiated in this session, mapped to the names they were bound to. Objects get added to this table with the *instPipe* and *copyPipe* commands. When a pipe is created, it will create stages, which involve calls to *instStage*. The newly created stages are saved in the *Stage Table*, shown in Table 5.5. Calls to *swapStage*, which happen when changes to the source code have prompted LiveCompiler to create new shared object files, causes changes to this table as well.

5.4.2.3 Simulation Updates

The *Object Library Table* (Table 5.3), *Pipeline Table* (Table 5.4), and *Stage Table* (Table 5.5) make it possible to quickly and automatically update the simulation as the user edits the source.

LiveCompiler sends metadata to LiveSim to tell it when it has compiled updates to stage and testbench objects. The metadata takes the form shown in Table 5.6. It contains a mapping showing where the updated object for a given source path can be found.

Table 5.6: Update metadata: when LiveCompiler recompiles a pipeline, stage, or testbench object it sends a metadata update to LiveSim, shown below.

Source Path	Object Path
/projects/src/toplevel.v#core	/livesim/objs/.../libc3.so#core
/projects/src/block.v#block	/livesim/objs/.../libc4.so#block

Using the *Object Library Table*, the *Pipeline Table*, and the *Stage Table*, LiveSim can determine which objects need to be updated. If one of those objects is the active simulation pipeline, this triggers a simulation reload, further discussed in Section 5.4.4.

5.4.2.4 A Note on Implementation

To implement this, it is necessary to divide the architecture into stages, and parse out the connections between them. In Verilog we have had reasonable performance by just designating every module instantiation as its own stage. LiveSim could also be configured to designate only some instances as their own stage by marking them with a configuration file or a custom comment flag. A potential advantage would be that grouping a larger block of code into a stage could allow for more optimization within that block.

The compiler must also be modified so that it will build stages as hot loadable libraries. Compilers like Verilator, which already output C++ based on templates, could be easily manipulated to that end. HDLs with non-native backends like Chisel [29] and PyMTL [59], generally want to run simulations natively for performance reasons, hence why Verilator is generally used. These languages would need to integrate their compiler into an incremental compilation flow which would entail generating compile-able Verilog for parts of an architecture, regenerating shared object files from those fragments, and patching them into the simulation.

5.4.3 LiveParser and LiveCompiler

Being able to quickly translate changes in code to updated simulation results is the primary function of a live programming backend. The first step in this flow for LiveSim is the LiveParser.

The LiveParser does not fully parse the HDL source code, it is only designed to identify which stage the change in code took place in, and confirm that actual behavior was changed, not just comments or spacing. LiveParser then extracts those sections of the codebase and sends *only those* to LiveCompiler for re-compilation.

For Verilog, recompiling a stage requires recompiling not just the modules that the stage instantiates, but the instantiation statement itself, in order to retrieve any compile-time parameters. This may also require recompiling pre-processor directives if the LiveParser indicates code in those regions have been changed. Other newer HDLs like CHISEL and Pyrope that do type resolution also allow programmers to make local changes which affect the global state.

LiveParser divides the code into regions based on the module structure, and the locations of pre-processor directives. Once it has confirmed a behavioral change in a code region, it must identify every region of the code base that *may* have been affected by those changes. For Verilog, if this is just a change within a module, the affected parts of the system are the instances which instantiate it and those further up in the hierarchy. For pre-processor directives, this could affect any code “below” the affected lines in the source code, thus much more will have to be recompiled.

When deploying LiveSim for use by developers, compilation should happen on a different execution thread, or even a different server than the simulation and editor interface so the user experience is minimally affected by the compilation overhead. The compiler may end up re-compiling code only to determine that many of those modules were ultimately not changed. Whenever it compiles a new version of a stage, pipeline, or testbench, it compares it against a cached copy to determine if this version needs to be swapped into the simulation. These decisions must take place off of the editor and simulation execution threads.

Some newer HDLs, like Chisel, employ type resolution. Compilation can be sped up by keeping the old type state in memory, but allowing it to be updated based on new code. This can significantly reduce the number of resolution cycles needed on subsequent compilations.

5.4.4 Fast Reloading

Once the simulation has been patched with the latest changes, the results to the user must be updated. The algorithm LiveSim uses is designed to provide a fast estimate of the updated simulation, and check/refine that estimate in the background, updating if necessary. The reloading mechanism is also tuned for what we refer to as the “edit-run-debug” loop, where the developer edits a line of code, runs the simulation, and debugs the result. In other words, the reload algorithm is tuned to the use case of fixing an error observed in a simulation.

LiveSim first reloads the checkpoint that is closest to 10K cycles before the stopping point (this parameter is tunable). From the literature we have found on the topic of hardware

Table 5.7: If the register names do not completely match from version to version, transformation rules are applied to provide a deterministic way to load checkpoints from the old version to the new one.

Scenario	Action
Register created	Initialize to 0 (or other value)
Register deleted	Ignore data from checkpoint
Single Register renamed	Map old-name to new-name in the checkpoint

debugging [69] and talking with experts in the area, this distance in cycles between bug and detection is often not that large, rarely more than a few thousand cycles. LiveSim reloads the state from that checkpoint and reruns the simulation. This result is reported immediately to the user as an estimate of the correct, updated simulation result.

The estimate may be inaccurate because the changes to the pipeline or testbench behavior which caused the update may also cause the system state to diverge at an earlier point in the simulation. To check for this we duplicate the newly patched pipeline and/or testbench and rerun the earlier checkpoints in parallel, to see if the states diverge at an earlier point in the session history. If so, update the final results as necessary. This allows LiveSim to provide fast updates to the user and hide as much of the computational effort on the backend as possible.

5.4.5 Reloading Rules

A checkpoint consists of the entire state of the pipeline object. In some cases, when the user makes changes to the system, they may add, remove, or rename registers, making it impossible to blindly transfer checkpoints between iterations of the same system.

Table 5.8: The *Register Transform History Table* lists the changes to an architecture's register topology require to translate the system's state from one version to the next. It is structured to allow for branching.

Version	Operations	Parent
1.1	create newR	null
1.2	create newR1 rename someR, newR	1.1
1.3	delete otherR rename newR1, myR1	1.2
1.3a	delete newR	1.2

LiveSim uses standardized rules to enable patched pipelines to deterministically load checkpoints from previous versions of the system, shown in Table 5.7. Since updates should happen frequently, ideally once per second as the user types, these cases can handle a large percent of the update batches without user intervention. If there have been too many changes between version for LiveSim to unambiguously determine the register mapping between two checkpoint versions, such as if multiple registers were renamed, then it will make its best guess based on the similarities of names and types. The user can manually edit the *Register Transform History* if the mapping is incorrect.

The *Register Transform History* maps the transformations applied to the architecture's registers in order to translate the checkpoints of one version into those of another. An example of such a history is shown in Table 5.8. The table is designed to support branching so that developers are not limited to a linear sequence of changes when exploring the design space.

5.4.6 Checkpoints and Memory Usage

The more checkpoints LiveSim has for the unit under test, the more quickly it can provide updated and verified simulation results as it has more points to work with. Each checkpoint requires memory, however, and when the checkpoint memory footprint gets too large it will be necessary to drop checkpoints as new ones are continually added. The cost of having checkpoints more sparsely laid out is that LiveSim will have to traverse longer deltas between them to determine when verifying the new version against the old.

To date, there has not been the opportunity to run a full test of the LiveSim pipeline to evaluate optimal sizes and allocate/deallocation strategies for LiveSim checkpoints.

5.4.7 Checkpoint Consistency Verification

One thing to note is that after a code change, it is possible that the checkpoints generated with the old version of the code may be invalid. That could be caused by a divergence in the execution, *i.e.*, a different condition was triggered somewhere throughout the simulation, or simply because the logic of the simulator was reworked. In any case, since LiveSim starts the simulation from a saved checkpoint, it is necessary to make sure that the checkpoint is a valid state for the given input set of the simulation. This requires re-running the simulation from the beginning, which is the opposite of what we are trying to achieve.

However, this consistency verification can be done from checkpoint to checkpoint, *i.e.*, it is possible to run the simulation starting from each of the checkpoints prior to the current cycle, finishing at the next checkpoint,[1] and then verify that the states match.

Since each checkpoint is now independent of other results, this operation can be easily made parallel and can scale to a large number of cores (as many as checkpoints before the current cycle). In general, if n cores will be used, the best strategy would be to divide the whole simulation into up to $n-1$ parts with roughly the same number of checkpoints in each. The last core is assigned the reload from the last checkpoint and executes up to the current simulation cycle. For instance, for 1B cycles in a 4 core, 3 cores will execute roughly 333M cycles, while the remaining core will execute the cycles from the last checkpoint up to the last cycle, feeding those results back into the debugging process. Figure 5.4 illustrates how the consistency verification works. In cases where the checkpoints are not consistent, this approach allows for identifying at which checkpoint the divergence occurred, which may also be useful for debugging.

5.5 Setup

In this section, we discuss the evaluation setup for LiveSim, the benchmarks used in the evaluation, and the infrastructure used for both LiveSim and Verilator [17]. We also discuss optimization options and how they were selected, since this ended up having an important impact on the results observed.

[1]Technically, any checkpoint after the starting checkpoint.

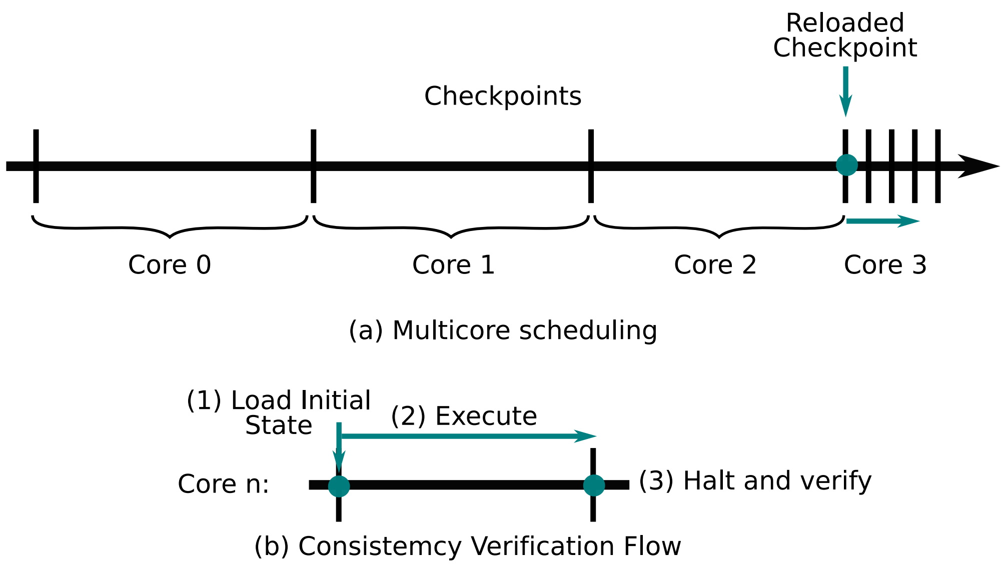


Figure 5.4: After a code change, it is important to verify the consistency across the checkpoints. This can be done in parallel and is easily scalable to a large number of cores. (Picture credit: R. Possignolo *publication pending*)

All evaluations were performed in an Archlinux 4.3.3-3 server, with a Intel i7-6700K, 32GB of ram. We used Verilator version 3.9 [17], and g++ 7.3 for all the simulations.

5.5.1 Benchmarking Architecture

The goal of the LiveSim project is to provide a live development environment which can scale up to even large digital architectures. To evaluate this, we use a partitioned global address space (PGAS) of RISC-V cores, in sizes: 1×1, 2×2, 4×4, 8×8, and 16×16.

Each PGAS node is a 5-stage RISC-V RV64i core with 32K internal memory. The nodes are arranged in a mesh, with each core having a memory address mapped section of the global address space. The address decides whether the memory operation is local or remote.

This architecture resembles Parallella [23] and Celerity [26] which are also a multicores that don't employ cache coherence.

The way this architecture was designed, each of the five stages of the RISC-V cores is placed in a module, which is instantiated by a single top-level parent, which is also its own module. LiveSim places each of these modules in its own shared library. Thus, all of the PGAS simulations are made up of seven shared libraries: 5 for the stages, 1 of the top-level which instantiates them, and 1 for the testbench. These libraries are used to create instances of the objects they contain, so for the 16×16 PGAS, there will be 6 instances for each of the 256 nodes in the mesh.

5.5.2 A Note on Compiler Configuration

Both LiveSim and Verilator generate C++, all testbenches were compiled with g++ using the same O2 optimization flag.

Verilator was unable to produce a C++ implementation for the 16×16 PGAS even after 24 hours of runtime. We tried many Verilator options like trying to reduce optimization, "output-split" manual selection, and even clang instead of g++. In all the cases, we could not finish the compilation in less than 24 hours.

5.6 Evaluation

The evaluation is divided in three main sections. We start by evaluating the Hot Reload speed and compilation speed. Then, we proceed to compare LiveSim against Verilator to

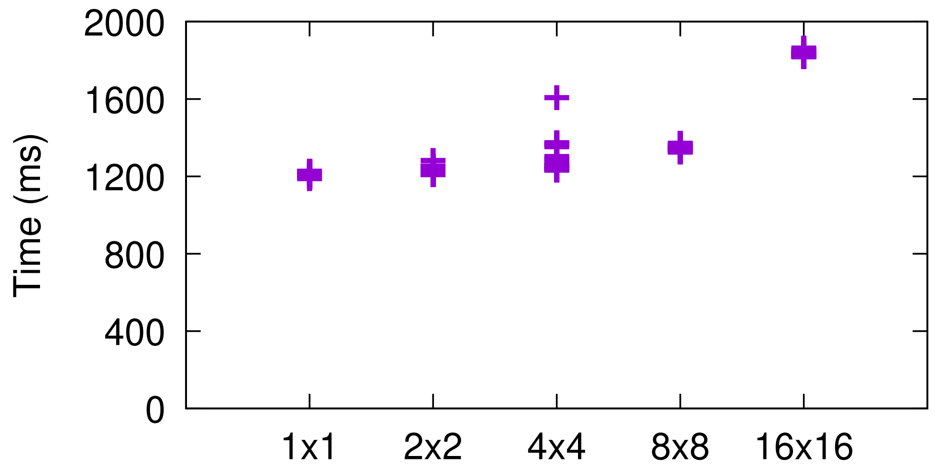


Figure 5.5: LiveSim can hot reload in less than 2 seconds from small PGAS with 1 node to large mesh with 256 PGAS multicore (16×16). Even in the case where a stage was duplicated hunderds of times in the design, the latency required to update the code for all instances of that stage is dominated by the parsing and compilation costs, which remain constant.

compare the simulation speed with an without Hot Reload. We conclude by quickly evaluating the checkpointing overheads.

5.6.1 Hot Reloading

To evaluate the effectiveness of LiveSim hot reload, we introduce bugs found during development in all the pipeline stages of the design. For each bug, we measure the time from the moment that the file is saved until the simulator is reloaded with the new code. This includes parsing, compilation, and reloading state on the simulator.

Figure 5.5 shows the main latency of the ERD loop for each size of the PGAS. 1×1 is a mesh with a single PGAS core, the 16×16 is a mesh with 256 PGAS. The first observation is that we have an ERD loop of less than 2 seconds in all the cases, even for the 256 PGAS where 256 pipeline stages are reloaded in the ERD loop. This is because the PGAS meshes are made by replicating the same RISC-V core, making a change to one core necessitates doing a swap on every core in the mesh.

A second observation is that replacing one or more pipeline stages has a small impact in the ERD loop, despite the fact that the number of stages being swapped is growing exponentially. This is due to the fact that the latency of the ERD loop is dominated by other parts of the flow, specifically LiveParser and LiveCompiler, which only happen once regardless of the size of the mesh.

A third observation is that there is very little variation independent of the bug or pipeline stage being handled. All these bugs affected a single pipeline stage. Not shown in the figure, but if two pipeline stages are affected, the time increase is linear.

Since LiveParser only has to do a minimal amount of work, its execution time was negligible for all benchmarks, less than 20ms.

This is not so surprising considering the process of hot patching a stage in detail. Once we have the shared library compiled and loaded into memory, a process which only has to be done once, LiveSim calls a method from the library which creates the new stage object, and copies the register values from the old one to the new one (taking into account any which have been added, removed, or renamed). For a single core, the size of all their internal memories (2×32 KB) and the register file (32×64 bits) comes to under 100 KB. The cost of copying that, even 256 times, is still eclipsed by other parts of the incremental compilation process.

Table 5.9: Compilation time for LiveSim and Verilator. Hot Reload can swap a module in under 2 seconds. Even full compilation with LiveSim is faster than with Verilator.

	1×1	2×2	4×4	8×8	16×16
LiveSim Hot Reload	1.5	1.5	1.5	1.6	2
LiveSim Full	4.9	4.7	4.8	15.6	176
Verilator	8	14	63	327	NA

To appreciate the LiveSim Hot Reload speed, we compare it against Verilator which is used in many HDL flows. Table 5.9 shows the compilation times for LiveSim and Verilator. As previously stated, LiveSim Hot Reload achieves a ERD of less than 2 seconds. A clean full compilation with LiveSim is slower. It goes from 4.9 seconds in a trivial single core to 176 seconds for a 256 PGAS. Verilator is always slower compiling, we were not able to compile the 256 PGAS in Verilator even though we tried clang, different verilator options, and even removed compile optimizations. In summary, for full compilation LiveSim has a higher starting overhead to partition the design, but it scales better because it shares code between pipeline stages.

5.6.2 Simulation Efficiency

LiveSim Hot Reload is very fast at reloading a stage, and it is also fast compiling large designs from scratch. This section provides insights on the LiveSim simulation speed.

Figure 5.6 shows the overall simulation speed of LiveSim and Verilator when simulating the PGAS designs of various sizes. The Verilog implementations were obtained by

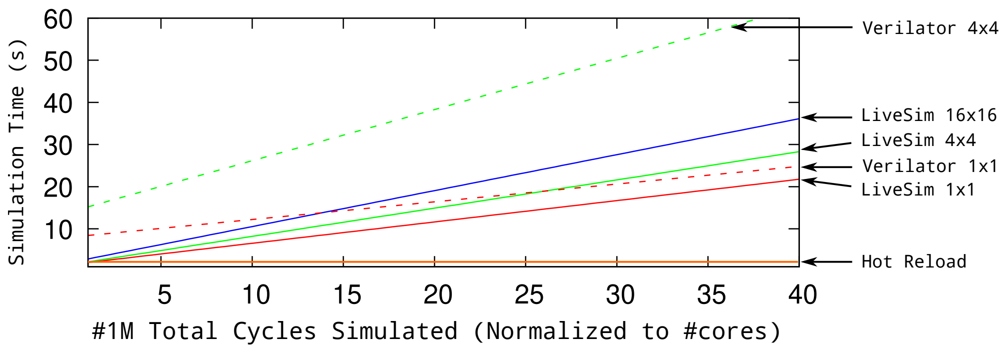


Figure 5.6: Even without checkpoints LiveSim has a faster starting (ERD) and a faster for large PGAS.

compiling the Pyrope PGAS source code to Verilog, and simulating with Verilator. LiveSim was working directly from Pyrope. In this plot, we benefit Verilator because we assume that we do not have checkpoints to replay from. In all the cases, we start from the beginning. In reality, LiveSim only needs to start from the last checkpoint.

The x-axis shows the total number of cycles simulated. In a 4×4 PGAS, it is the wall clock cycle multiplied by 16. The y-axis shows the time to reach that point in the simulation. Since LiveSim ERD is less than 2 cycles, all the LiveSim lines start at 2 seconds. The Verilator lines start at different points depending on the compilation time shown in Table 5.9. The slope of the lines is defined by the simulation speed. The faster the simulation, the lower the slope. Table 5.10 includes all the simulation speeds.

LiveSim has a faster starting point for the 4×4 PGAS, Verilator needs 63 seconds to start while just 2 seconds are needed for LiveSim. LiveSim also has a smaller slope because it is faster. While Verilator achieves 718 KHz, LiveSim runs at 1223 KHz. This is over 1.7× speedup. As a result, to run something like 10 million cycles, Verilator needs 257 seconds, and LiveSim needs 109 seconds. Although Verilator is somewhat faster on the smaller architectures,

Table 5.10: As the architecture grows in size, Verilator's cache performance degrades significantly.

	PGAS 1×1		PGAS 2×2		PGAS 4×4		PGAS 8×8		PGAS 16×16	
	LiveSim	Verilator	LiveSim	Verilator	LiveSim	Verilator	LiveSim	Verilator	LiveSim	Verilator
KHz	1974	2378	1957	2351	1492	823	1223	718	1172	NA
IPC	2.50	2.86	2.47	2.71	1.94	0.96	1.62	0.80	1.24	NA
I$ MPKI	0.25	0.01	0.02	8.45	0.01	20.57	0.01	24.37	0.01	NA
D$ MPKI	0.96	0.01	10.78	0.01	36.42	84.04	39.09	42.99	48.08	NA
BR MPKI	1.54	0.01	1.33	0.01	2.83	0.01	3.62	0.29	4.27	NA

even then it can only outperform LiveSim when a very large number of cycles has been run due to its longer compile time. For the 1×1 PGAS, Verilator only passes LiveSim after running 76 million cycles. As the design gets more complex, LiveSim gets faster even when no checkpoints are available.

To understand the simulation speed difference between Verilator and LiveSim we gathered performance counter statistics. Table 5.10 shows a breakdown of performance statistics for each PGAS size. Here we can see the source of the performance divergence observed in Figure 5.6. Verilator's cache performance, particularly for the I-cache, degrades significantly for the PGAS 4×4 and larger, causing a drop in IPC. For small designs, the Verilator design fits in the host instruction cache, but as more cores are added, the I$ pressure makes the simulator very slow.

Although not a technique planned for this work, LiveSim also improves data cache performance. The LiveSim simulator inserts more "if" statements or other conditional blocks for muxes than Verilator. The result is an increase in branches and a decrease on cache conflicts.

This optimization does not seem to yield benefits for small scale designs because they fit in the data cache. As the design gets larger, however, branch miss prediction rate increases, but the rate of increase of data cache Misses Per Thousand Instructions (MPKI) is slower than in Verilator.

The explanation for this trend is due to the difference in LiveSim and Verilator's internals. Verilator inlines nearly everything, creating massive C++ functions, while LiveSim splits the logic into separate shared object libraries shared between stage instances. Verilator's architecture is advantageous for small designs, but as the system scales, so does the simulation executable size which begins to negatively affect performance.

Verilator's poor I-cache performance is the primary driver in its performance degradation. Inlining the entire system is more efficient for small architectures because of superior data locality, but once the simulation is large enough to saturate the caches, then that quickly becomes a bottleneck.

LiveSim has a better performance scalability as a result of the improvements like "if" insertion and mostly due to instruction cache MPKI reduction. The result is that going from 1 core to 256 (16×16) LiveSim just has a 40% performance degradation. Verilator is not able to compile such large design, but going from 1 to 64 PGAs has 70% performance degration. As a reference, LiveSim has just a 25% degradation for this data point.

The result is that LiveSim not only has a 2 second ERD cycle, it also compiles faster and has faster simulation speeds for large designs.

5.6.3 Checkpointing

One interesting characteristic from LiveSim is that it has checkpointing capability. As such, it can execute many checkpoints in parallel when there is a code change. The evaluation has ignored the checkpointed capacity because there is no platform to compare against. Even more complicated is that most of the bugs introduced are triggered early in the replay. If we had more complicated bugs that require thousands of cycles for being reproduced, we should be able to evaluate this feature more fairly.

For this evaluation, we do evaluate the overhead of taking checkpoints. The overhead is dependent on the PGAS, but in all the cases it remains in the 10-20% range. This includes the checkpoint creation as the simulation runs. Without this overhead, LiveSim would be nearly as fast as Verilator even for the simple 1×1 PGAS.

To summarize, checkpoints have a 10-20% simulation speed overhead, and that allows for avoiding the execution or checking in parallel the number of checkpoints for a code change. If checkpointing is not used it can be disabled. Hot Reload is still available, but the simulation must run from the beginning.

5.7 Conclusion

This section discusses LiveSim, a live development environment for HDLs built on top of the Pyrope compiler. LiveSim requires a sizable amount of infrastructure, but as the evaluation shows it has significant performance benefits compared to traditional flows. More importantly, it allows for incremental compilation and hot reloading, which is not currently available to hardware developers.

Hardware development is plagued by notoriously long compile and simulation times which forms a bottleneck to productivity as developers spend much of their time waiting for feedback. There are a variety of software design tools and libraries which have embraced the idea that faster feedback is better for developers and have implemented a variety of features aimed at providing it. We believe its time for hardware design tools to do so as well.

This work LiveSim includes details on how to implement a Hot Reload efficiently. We also show that current popular tools like Verilator have scalability problems due to instruction caches. We show that the same techniques used to create libraries for Hot Reloading benefit the scalability.

The evaluation shows that LiveSim compiles faster than Verilator, and it also has a faster simulation time for larger designs. Maybe even more important, we demonstrate a 2 second edit-run-debug (ERD) loop even for large scale SoCs.

LiveSim hot reload is a new way to interact with hardware simulators. While current flows are slow, and require the user to be very careful about the code changes when debugging, hot reload opens the opportunity to create new ways of thinking. For example, since hot reload is fast. The designer can insert print statements and replay from any given point with very low overhead. Similarly, the designer can explore “what if” scenarios much more quickly than with current tools. Another interesting feature for traditional flows, hot reload allows for hiding costly initialization. If there is a slow reset and program load in a design, hot reload can start afterwards.

Chapter 6

Conclusion and Future Work

All the world's a stage, and all men and women merely players; they have their exits and their entrances, and one man in his time plays many parts,

William Shakespeare, *As You Like It*

The Pyrope project was started with the goal of improving digital architecture development tools by leveraging fluid pipelines. Although the project is still ongoing, this paper focuses on the progress made so far. An initial version of the Pyrope programming language was developed, which incorporates an Actor Model based paradigm, discussed in Chapter 3 to model fluid behavior implicitly, without sacrificing control. As discussed in Chapter 4, the Pyrope compiler can implement fluid pipeline transformations as part of the compilation flow. Though these mechanics are still in their early stages, Chapter 4 also shows several ways they can be leveraged to aid in the development process. Finally, Chapter 5 shows how the Pyrope compiler can be integrated into a live programming environment, and how incremental compilation and simulation techniques can be used to improve developer productivity. Chapter 5 also shows that the Pyrope compiler is faster than Verilator [18], and shows how the optimizations that enable this can be integrated into current HDL design flows.

Nonetheless, the Pyrope project is still in its early stages. I will divide the future work in this chapter into three different sections: future work on the Pyrope language 6.1, on the Pyrope compiler and toolchain 6.2, and on pipeline transformations 6.3.

6.1 The Pyrope Language

All the work done thus far was done with a minimal set of language features, we in the MASC lab refer to that source code as Pyrope-alpha. Besides the Abort paradigm, discussed in Chapter 3.1, the language had few other features other than basic operators, signed and unsigned integer types, and functions. In the future, we at the MASC lab would like for Pyrope to incorporate a lot of modern programming language features that many HDLs lack, such as class types and polymorphism, built in list and map data-types, and global type inference. We also envision a more robust type system which can better prevent errors like overflows and underflows which are a common source of bugs. We believe these can be integrated into the language without sacrificing performance and control, although the specifics have not all been fully flushed out.

The Pyrope language also lacks a lot of semantics needed to write testbenches, including loops, requiring all testbenches up to this point be written in C++. Although C++

testbenches are a useful tool, we believe being able to write testbenches in native Pyrope will allow developers to better test, and leverage, the fluid nature of such systems.

6.2 The Pyrope Compiler and Toolchain

There is still a lot of work to be done in bringing many of the ideas discussed in this paper from proof-of-concept to reality.

A new version of the Pyrope compiler is being developed on top of MASC's LGraph Infrastructure, which is a fast, scalable, serializable graph backend which is designed for digital architecture development. We are also working toward to goal of bringing a fully functional LiveSim server online, and also hope to implement it as a collaborative development platform as well.

Once up and running, we will need to run user studies and partner with companies in industry as we continue fine tune the toolchain. LiveSim has the potential to change the general workflow of architecture debugging/verification. Parallel checkpoint verification and checkpoint version transformations, discussed in Section 5.4.4, properly leveraged, could mean that developers will rarely have to run a simulation from the beginning. Combined with incremental compilation could allow developers to experiment with "what-ifs", leaning more heavily on their tools.

Finally, on the long term we hope to integrate Pyrope, through LGraph to LLVM, connecting it with a much larger developer ecosystem.

6.3 Pipeline Transformations

Research thus far has only focused on the *merge* operation, though LI literature defines more transformations, including a *split* operation and *recycling* and *retiming* which allow for logic to be moved between stages without changing changing the number of register blocks. A long term goal for the Pyrope project is a system which can leverage a wide variety of pipeline transformations to allow developers to fine tune the timing behavior of the system without changing the source code. Doing so could be a real game changer, as we believe, especially as architectures continue to grow in size. On the long-term one of the tools we envision creating is one which could full leverage *merge* and *split* (to increase and decrease the stage count), and *recycling* and *retiming* (to move logic between stages), to create a more balanced pipeline in an automated flow than could be done by hand.

Fluid Formal Verification, discussed in Section 4.6, has only briefly been studied, and has not been published off of. Future work could look at leveraging it to allow variations of the same architecture, or different versions of the same architecture to be verified against each other formally with little additional developer effort. We believe both that there is a lot to explore in this area, and that doing so will be important to addressing the additional verification challenges discussed in Section 2.1.3.

6.4 Final Thoughts

The Pyrope Project proposes a radically different design flow than the traditional approach to digital architecture development, but we believe these differences address inherent issues with the current digital design flow. Pyrope is designed to leverage fluid pipelines and its unique programming paradigm to allow a higher level of control of architectures, without leaving the RTL level, or sacrificing control and tunability.

Although our focus remains on micro-architectures, we believe Pyrope's paradigm could be applicable to other areas, such as networking and Internet-of-Things (IoT) architectures. In many ways, fluid pipelines can be described as a synthesizable, distributed system. I personally believe, on the long term, this model can serve as a way to more closely integrate hardware and software development in general.

Bibliography

[1] Akka: A concurrency framework for java and scala. http://akka.io/. Accessed: 2017-05-25.

[2] C++ gets squiggles! https://blogs.msdn.microsoft.com/vcblog/2009/06/01/c-gets-squiggles/. Accessed: 2018-04-06.

[3] Cal actor language homepage. https://ptolemy.berkeley.edu/projects/embedded/caltrop/language.html. Accessed: 2018-05-19.

[4] Colt - code orchestra livecoding tool. https://en.wikipedia.org/wiki/COLT_(software). Accessed: 2018-04-06.

[5] Compileonsave - netbeans wiki. https://wiki.netbeans.org/CompileOnSave. Accessed: 2018-04-06.

[6] The esterel programming language. http://www-sop.inria.fr/meije/esterel/esterel-eng.html. Accessed: 2017-05-08.

[7] Introducing hot reloading - react native. https://facebook.github.io/react-native/blog/2016/03/24/introducing-hot-reloading.html. Accessed: 2018-04-06.

[8] Orbit: a framework for writing distributed systems using virtual actors. https://github.com/orbit/orbit/wiki. Accessed: 2017-04-22.

[9] Picorv32: A size optimized risc-v cpu. https://github.com/cliffordwolf/picorv32. Accessed: 2017-05-25.

[10] Pulp platform: Parallel ultra low-power platform. http://www.pulp-platform.org/. Accessed: 2017-11-20.

[11] Pulsar, a python concurrency framework. http://quantmind.github.io/pulsar/. Accessed: 2017-04-22.

[12] Pykka: A python implementation of the actor model. http://ptolemy.eecs.berkeley.edu/. Accessed: 2017-05-25.

[13] Ri5cy: A small, 4-stage risc-v core. https://github.com/pulp-platform/riscv. Accessed: 2017-11-20.

[14] Spike: An open-source risc-v emulator. https://github.com/riscv/riscv-isa-sim. Accessed: 2017-11-19.

[15] Synopsys vcs simulator. https://www.synopsys.com/verification/simulation/vcs.html. Accessed: 2018-05-19.

[16] System-c homepage. http://www.accellera.org/downloads/standards/systemc. Accessed: 2018-05-19.

[17] Verilator 3.9, open source tool for verilog hdl simulation. Online Webpage. https://www.veripool.org/wiki/verilator.

[18] Verilator: A fast and free verilog hdl simulator. https://www.veripool.org/wiki/verilator. Accessed: 2017-11-20.

[19] Vscale. https://github.com/ucb-bar/vscale. Accessed: 2017-4-4.

[20] Zero-riscy: A small, 2-stage core derived from ri5cy. https://github.com/pulp-platform/zero-riscy. Accessed: 2017-11-20.

[21] Labview actor framework "white paper". https://forums.ni.com/ni/attachments/ni/7301/130/1/Using2011.

[22] The 2016 Wilson Research Group Functional Verification Study, 2016.

[23] Adapteva Inc. Parallella-1.x reference manual, 2014.

[24] Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(01):1–72, 1997.

[25] Tariq B. Ahmad and Maciej J. Ciesielski. Fast sta prediction-based gate-level timing simulation. *2014 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2014.

[26] Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Atieh Lotfi, Julian Puscar, Anuj Rao, et al. Celerity: An open-source RISC-V tiered accelerator fabric. In *A Symposium on High Performance Chips (Hot Chips 29)*, Aug. 2017.

[27] Saleh Alhazbi and Aman Jantan. Multi-level mediator-based technique for classes hot swapping in java applications. In *Information and Communication Technologies, 2006. ICTTA'06. 2nd*, volume 2, pages 2889–2892. IEEE, 2006.

[28] Ehsan K. Ardestani and Jose Renau. ESESC: A fast multicore simulator using time-based sampling. In *High Performance Computer Architecture, Proceedings of the IEEE 19th International Symposium on*, HPCA'13, pages 448–459, Washington, DC, USA, Feb. 2013. IEEE Computer Society.

[29] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Design Automation Conference, Proceedings of the 49th Annual*, DAC '12, pages 1216–1225, New York, NY, USA, Jun. 2012. ACM.

[30] Somnath Banerjee and Tushar Gupta. Optimized simulation acceleration with partial testbench evaluation. In *2014 15th International Microprocessor Test and Verification Workshop*, pages 22–27, Dec 2014.

[31] Raymond E. Barber and Henry C. Lucas Jr. System response time operator productivity, and job satisfaction. *Communications of the ACM*, 26(11):972–986, 1983.

[32] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[33] David Broman, Christopher Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. Determinate composition of fmus for co-simulation. *13th International Conference on Embedded Software (EMSOFT)*, 2013.

[34] Dmitry E. Bufistov, Jordi Cortadella, Marc Galceran-Oms, Jorge Julvez, and Mike Kishinevsky. Retiming and recycling for elastic systems with early evaluation. In *Design Automation Conference, Proceedings of the 46th ACM/IEEE*, DAC'09, pages 288–291. IEEE, Jul. 2009.

[35] Andrew Butterfield and Jim Woodcock. A "hardware compiler" semantics for handel-c. *Electronic Notes in Theoretical Computer Science*, 2006.

[36] Luca P. Carloni, Kenneth L. McMillan, Alexander Saldanha, and Alberto L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency-insensitive design. In *Computer-Aided Design, Digest of Technical Papers of the IEEE/ACM International Conference on*, ICCAD'99, pages 309–315. IEEE, Nov. 1999.

[37] Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In *Design Automation Conference, Proceedings of the 37th*, DAC'00, pages 361–367, New York, NY, USA, Jun. 2000. ACM.

[38] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A. Patterson, and Krste Asanović. Boom v2: an open-source out-of-order risc-v core. Technical Report UCB/EECS-2017-157, EECS Department, University of California, Berkeley, Sep 2017.

[39] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. A pythonic approach for rapid hardware prototyping and instrumentation. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pages 1–7. IEEE, 2017.

[40] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-synchronous kahn networks: A relaxed model of synchrony for real-time systems. *Symposium on Principles of Programming Languages*, 2006.

[41] Jim Dabrowski and Ethan V. Munson. 40 years of searching for the best computer system response time. *Interacting with Computers*, 23(5):555–564, 2011.

[42] Stephen A Edwards. High-level synthesis from the synchronous language esterel. In *IWLS*, pages 401–406. Citeseer, 2002.

[43] Stephen A. Edwards, Richard Townsend, and Martha A. Kim. Compositional dataflow circuits. *Conference of Formal Methods and Models of Codesign (MEMOCODE)*, 2017.

[44] Adrian Evans, Allan Silburt, Gary Vrckovnik, Thane Brown, Mario Dufresne, Geoffrey Hall, Tung Ho, and Ying Liu. Functional verification of large asics. In *Proceedings of the 35th Annual Design Automation Conference*, DAC '98, pages 650–655, New York, NY, USA, 1998. ACM.

[45] M. Galceran-Oms, J. Cortadella, and M. Kishinevsky. Symbolic performance analysis of elastic systems. In *Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on*, pages 778 –785, Nov. 2010.

[46] Ilya Ganusov, Henri Fraisse, Aaron Ng, Rafael Possignolo, and Sabya Das. Automated extra pipeline analysis of applications mapped to xilinx ultrascale+ fpgas. *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, 2016.

[47] Dave Griffiths and Gabor Papp. Fluxus programming environment. http://www.pawfal.org/fluxus/.

[48] T.S. et. al. Harris. Techniques for li-bdn synthesis for hybrid micro-architectural simulation. *ICCD'11*, October 2011.

[49] Carl Hewitt, Peter Biship, and Steiger. A universal modular actor formalism for artificial intelligence. *Proceedings of the 3rd international joint conference on Artificial intelligence*, 1973.

[50] Syed Iqbal, Mitul Soni, and Gourav Kapoor. Timing closure in multi-level partitioned socs. https://www.eetimes.com/document.asp?doc_id=1274615. Accessed: 2018-05-20.

[51] Jorge Julvez, Jordi Cortadella, and Mike Kishinevsky. Performance analysis of concurrent systems with early evaluation. In *Computer-Aided Design, Proceedings of the IEEE/ACM International Conference on*, ICCAD'06, pages 448–455, New York, NY, USA, Nov. 2006. ACM.

[52] Gilles Kahn. The semantics of a simple language for parallel processing. *Information Processing*, 1974.

[53] Donggyu Kim, Adam Izraelevitz, Christopher Celio, Hokeun Kim, Brian Zimmer, Yunsup Lee, Jonathan Bachrach, and Krste Asanović. Strober: Fast and accurate sample-based energy simulation for arbitrary RTL. In *Computer Architecture, Proceedings of the 43rd International Symposium on*, ISCA'16, pages 128–139, Piscataway, NJ, USA, Jun. 2016. IEEE Press.

[54] Dusung Kim, Maciej Ciesielski, Kyuho Shim, and Seiyang Yang. Temporal parallel simulation: A fast gate-level hdl simulation using higher level models. In *2011 Design, Automation Test in Europe*, 2011.

[55] Dusung Kim, Maciej Ciesielski, and Seiyang Yang. Multes: Multilevel temporal-parallel event-driven simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2013.

[56] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.

[57] Geoffry N. Lambert. A comparative study of system response time on program developer productivity. *IBM Systems Journal*, 23(1):36–43, 1984.

[58] Maysam Lavasani. *Generating Irregular Data-Stream Accelerators (thesis)*. PhD thesis, University of Texas at Austin, 2015.

[59] Derek Lockhart, Gary Zibrat, and Christopher Batten. Pymtl: A unified framework for vertically integrated computer architecture research. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 280–292. IEEE, 2014.

[60] Colin Moock. *Essential ActionScript 3.0: ActionScript 3.0 Programming Fundamentals*. " O'Reilly Media, Inc.", 2007.

[61] Rishiyur Nikhil. Bluespec system verilog: efficient, correct RTL from high level specifications. *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, Jun. 2004.

[62] Daniel Pilaud, N Halbwachs, and JA Plaice. Lustre: A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, volume 178, page 188, 1987.

[63] Rafael T. Possignolo, Elnaz Ebrahimi, Haven Skinner, and Jose Renau. FluidPipelines: Elastic circuitry meets out-of-order execution. In *Computer Design, Proceedings of the 34th International Conference on*, ICCD'16, pages 233–240, Washington, DC, USA, Oct. 2016. IEEE Computer Society.

[64] Rafael T. Possignolo, Elnaz Ebrahimi, Haven Skinner, and Jose Renau. FluidPipelines: Elastic circuitry without throughput penalty. In *Logic Synthesis, Proceedings of the International Workshop on*, IWLS'16, Jun. 2016.

[65] Rafael T. Possignolo, Elnaz Ebrahimi, Haven Skinner, and Jose Renau. Automated pipeline transformations with Fluid Pipelines. In Reis Andre and Drechsler Rolf, editors, *Advanced Logic Synthesis*, pages 125–150. Springer, Cham, Switzerland, 2018.

[66] Rafael T. Possignolo and Jose Renau. LiveSynth: Towards an interactive synthesis flow. In *Design Automation Conference, Proceedings of the 53rd*, DAC'17, pages 74:1–74:6, New York, NY, USA, Jun. 2017. ACM.

[67] Hao Qian and Yangdong Deng. Accelerating rtl simulation with gpus. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 687–693, Nov 2011.

[68] Kyuho Shim, Youngrae Cho, Namdo Kim, Hyuncheol Baik, Kyungkuk Kim, Dusung Kim, Jaebum Kim, Byeongun Min, Kyumyung Choi, Maciej Ciesielski, and Seiyang Yang. A fast two-pass hdl simulation with on-demand dump. In *2008 Asia and South Pacific Design Automation Conference*, 2008.

[69] Sangeetha Sudhakrishnan, Liyang Su, and Jose Renau. Processor verification with hw-bughunt. In *ISQED '08: Proceedings of the 9th international symposium on Quality Electronic Design*, pages 224–229, Washington, DC, USA, Mar. 2008. IEEE Computer Society.

[70] Angela Sutton and Jeff Garrison. How to achieve timing-closure in high-end fpgas. https://www.eetimes.com/document.asp?doc_id=1274615. Accessed: 2018-05-20.

[71] Jose I. Villar, Jorge Juan, Manual J. Bellido, Julian Viejo, David Guerrero, and J. Decaluwe. Python as a hardware description language: A case study. In *2011 VII Southern Conference on Programmable Logic (SPL)*, pages 117–122, April 2011.

[72] Ge Wang and Perry R. Cook. Chuck: A concurrent, on-the-fly, audio programming language. *Proceedings of the International Computer Music Conference (ICMC)*, 2003.

[73] Clifford Wolf. Yosys open SYnthesis suite. http://www.clifford.at/yosys/, 2016. Online; accessed on 8 November 2018.

[74] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 84–95. IEEE, 2003.

[75] Craig Zilles and Gurindar Sohi. Master/slave speculative parallelization. *35th International Symposium on Micro-Architecture (MICRO)*, 2002.